Tool-Based Integration and Code Generation of Object Models

THESIS

Michael Ray Ashby, Capt, USAF

AFIT/GE/ENG/00M-02

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

20000815 188

Tool-Based Integration and Code Generation of Object Models

THESIS

Presented to the Faculty of the School of Engineering and Management

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Michael Ray Ashby, B.S. Electrical and Computer Engineering

Capt, USAF

March 2000

AFIT/GE/ENG/00M-02

Tool-Based Integration and Code Generation of Object Models

Michael Ray Ashby, B.S. Electrical and Computer Engineering

Capt, USAF

Approved:

Thomas C. Hartrum   _3 Mar 2000_

Dr. Thomas C. Hartrum   Date
Committee Chair

Robert P. Graham   _3 MAR 2000_

Maj. Robert P. Graham, Jr., Ph.D.   Date
Committee Member

Michael Talbert   _3 Mar 2000_

Maj. Michael L. Talbert, Ph.D.   Date
Committee Member

Acknowledgements

They that wait upon the Lord shall renew their strength; they shall mount up with wings as eagles; they shall run, and not be weary; and they shall walk and not faint.

Isaiah 40:31

There are many people that made it possible for me to complete this effort. I could never have finished this effort without the love and support of my wonderful wife, Wendy. She was always there supporting and helping me, even while being pregnant and experiencing dangerous complications. I am thankful for the patience of my two young sons, James and Kent, who had to endure a sick mother and an absent father. I also want to thank my parents, Gary and Millie Ashby, and my in-laws, Jack and Ann Mayson. My father always believed I could do the impossible, and somehow made me believe it also. My mother came and stayed with us during the final thesis stages, allowing me to complete my work, and my wife to regain her health. My mother- and father-in-law were constantly calling to provide encouragement and insure the safety of my family.

To my thesis advisor, Dr. Hartrum, and my committee members, Major Graham and Major Talbert, I am thankful not only for their encouragement and guidance, but also for their patience and flexibility while helping me to accomplish both my thesis work and taking care of my family. They always made sure that my family came first, and then my thesis.

Finally, I would like to thank the Lord. As always, He was the one that gave me the strength to complete this work and kept my family safe while I was doing it.

<div align="right">

Michael Ray Ashby

</div>

Table of Contents

List of Figures

AFIT/GE/ENG/00M-02

Abstract

Today many organizations are faced with multiple large legacy data stores in different formats and the need to use data from each data store with tools based on the other data stores' formats. This thesis presents a tool-based methodology for integrating object-oriented data models with automatic generation of code. The generated code defines a global data format, generates views of global data in individual integrated data formats, and parses data from individual formats to the global formats and from the global format to the individual formats. This allows for legacy data to be translated into the global format, and all future data to be entered in the global format. Once in the global format, the data may be exported to any of the integrated formats for use with the appropriate tools. The methodology is based on using formal methods and knowledge-based engineering techniques with a transformation system and object-oriented views. The methodology is demonstrated by a sample implementation of the integration tool being used to integrate data formats used by three different sensor-based, engagement-level simulation systems.

Tool-Based Integration and Code Generation of Object Models

I. Introduction

1.1 Background

The Air Force Research Laboratory (AFRL) uses many avionics simulation systems. The use of simulation systems reduces the effort, cost, and risk to personnel and equipment required for evaluating new techniques, strategies, and equipment. However, in the case of the AFRL avionics sensor simulation systems, the savings in cost and effort are greatly reduced due to a lack of interoperability between different simulation systems and the lack of user-friendly, time-saving tools. Each simulation system requires unique data stores, containing various scenario information obtained from multiple sources of varying formats, to support execution. The number and type of data stores required for each simulation system is different, and the tools for developing these data stores are unique for each simulation system and are time-, effort-, and knowledge-intensive. Furthermore, they are so user-unfriendly and menu intensive that users tend to avoid using them whenever possible. Therefore, if a scenario is developed and used for one simulation and then found desirable to be used for a different simulation, the user has to start over from scratch and use different tools to re-develop the scenario, even if the majority of the data is the same. This is an unacceptable burden in this age of decreasing manpower and resources.

1.2 General Problem

AFRL's problem is not a unique one. Legacy database systems and data stores of similar data in heterogeneous formats are commonplace. The problem is two-fold. First, schemas must be extracted from the heterogeneous data and then techniques must be developed to merge these schemas into a common data storage. The common data storage should be capable of producing the same information as each of the separate systems. This allows all future data obtained to be usable by tools based on any of the individual data stores. Second, a way must be provided to parse data from the local storage formats into

1

the common data store. This allows the legacy data to be converted to the global format with minimal effort, making it available for tools based on any of the data formats. The combined information should be able to be accessed by user-friendly interfaces and tools, displayed in multiple varying views, and able to be queried. In today's world of changing technologies and products, it would be very advantageous to be able to use this generic storage from multiple hardware and software platforms and products.

1.3 Initial Assessment of Past Effort

There have been and are many research efforts dealing with extracting schemas from heterogeneous data stores and merging the schemas. More specifically, several efforts have been completed and are ongoing to solve AFRL's simulation time, expense, and interoperability issues. AFRL is currently studying a concept known as Collaborative Engineering Real Time database CORrelation Tool (CERTCORT) [34]. This effort involves developing a database engine that can aid users in collecting information and then provide the information in various simulation formats. Currently CERTCORT is working with nine of AFRL's simulation systems.

AFRL has also sponsored several AFIT research efforts. Weber's thesis [38] deals with developing techniques to extract schemas from existing SUPPRESSOR and SWEG scenario files, implementing them as object oriented databases which can then be merged into a common database. Once this is accomplished, parsers can be developed which parse data from a current scenario file into the generic database and back out as a different simulation's scenario file. Colonese's thesis [6] develops methodologies and tools which, given specific simulation scenario object-oriented database schemas and data files, merges them into a generic database. Once information is in the generic database, database schemas and data files can be extracted for other simulation systems. The combination of Weber's and Colonese's theses demonstrates and provides methodologies which take data and schemas from specific simulation scenario files into an object-oriented generic database, and back out into other specific simulation scenario files. Stratton's thesis [35] evaluates user interface tools and provides an object-oriented database-based, user-friendly, agent-aided user interface for SUPPRESSOR. In Figure 1, the combined efforts of the three theses

2

are shown; solid lines indicate applications or implementations of the methodologies, and dashed lines represent possible implementations or extensions of the methodologies.
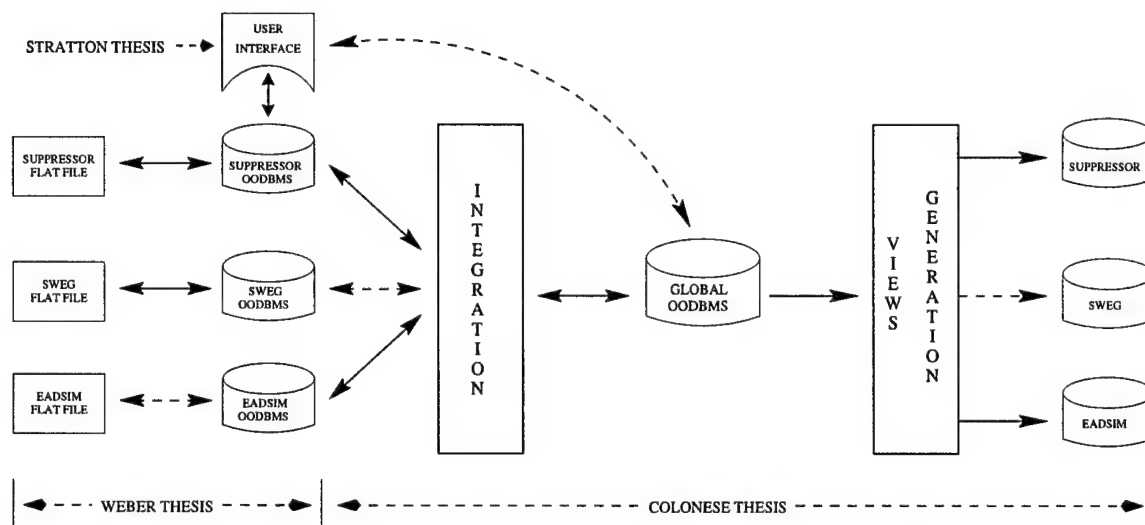
Figure 1.   Past Efforts in Support of CERTCORT

1.4   Specific Problem

The three theses presented above provide a strong basis for overcoming AFRL's interoperability problems. However, the methodologies focus on guiding the user through manual processes to achieve the desired results. Also, the tools developed and the demonstrations shown focus on using C++ and the ObjectStore® Object-Oriented Database Management System (OODBMS) (Stratton uses Java™ instead of C++).

The goal of this thesis was to apply formal methods and Knowledge Based Software Engineering (KBSE) tools and techniques to merging similar data and extracting the data out again by:

- Providing assistance with design decisions and partial automation in merging object-oriented schemas together.

- Providing automatic code generation, in multiple languages and data storage facilities, for schemas, parsers (for both into and out of specific and generic formats), and views.

3

1.5 Scope

This thesis does not focus on providing a fully capable, graphical user interface system. Given the interface development tools available today, Stratton's thesis, and some follow-on thesis work being done by McDonald [23], it should not be difficult to provide an appropriate user-interface once the underlying functionality is achieved. However, some graphical tools, which facilitate automation, are currently being used or available for formal methods and object-oriented modeling. These tools were evaluated and applied where applicable. The generation of user interfaces was generally considered low priority and outside the scope of the effort. Some user interfaces were minimally developed to allow the demonstration of the methodology.

This effort assumes object-oriented schemas with appropriate data stores are provided as input to and are the resulting output of the system. Sections 3.2 and 4.2 contain some discussion and recommendations for how to develop or extract these object-oriented models for non-object-oriented data stores. Further discussion and actually developing the object-oriented models were considered to be out of scope for this effort.

Although an object-oriented database was initially considered for use with the generic schema, other options were also explored. These options included using flat files, relational, or object-relational databases for storage, or even bypassing storage altogether and automatically generating tools for translating data from one simulation format to another. It was found that using an object-oriented generic schema facilitates the ability to integrate many different types of schemas. Since flexibility in the programing language and format of the local schema is a major goal of this effort, a choice of using an object-oriented database or some other object-oriented storage should be provided.

1.6 Approach

The following approach was followed for this effort:

1. Formal method techniques including areas and techniques that have been used to merge similar data and schemas were reviewed. An increased understanding of AFRL's simulation problem and currently proposed solutions was developed. Ca-

pabilities of various object-oriented, relational, and object-relational database management systems were explored.

2. Formal methods and KBSE tools and techniques were identified that could be used to:

   - Assist or automate integration of object-oriented schemas into a generic object-oriented schema.

   - Automate parsing data from the specific object-oriented schemas into the generic object-oriented schema and extracting data from the generic schema into the individual schemas.

   - Automate the generation of specific simulation views of the data in the generic schema.

3. Transformation systems and formally defined object-oriented views [15] were identified as the tools and techniques most applicable to integrating object-oriented schemas into a common format and facilitating code generation for definitions and tools using the local and global schemas. A tool-based methodology was developed to apply these tools and techniques. The methodology must be tool-based to achieve even a minimal amount of automation or code generation. Another advantage of the methodology being tool-based is that the tool can shield the integrators from the formal nature of the tools and techniques involved. This allows for integration to take place without the integrators requiring formal methods training. Chapter 5 presents ways to expand the capabilities and automation of the methodology by using more formal methods and requiring the integrators to provide formal inputs.

4. An implementation of the methodology's interactive tool was developed and used to demonstrate the methodology by integrating three battle simulation systems. Code was automatically generated in C++ and Java independent of any special formatting needed for use with an OODBMS or other tools. The required work to develop code generators for code in the proper format for an OODBMS and a Java Swing graphical tool were described, but not implemented due to time constraints.

1.7 Contributions

This effort developed many tools and techniques for use with integrating object models. A tool-based methodology was developed, which focuses the majority of the time, cost, and effort of integration into a reusable tool, and provides automated data transformations and code generation. A prototype Schema Integration Tool was developed and used to demonstrate the feasibility and capabilities of the methodology. A classification of heterogeneity conflicts and their resolutions was developed. This classification provides the capability and flexibility of classifications based on using SQL, without requiring the use of SQL or a database management system. Finally, a new object-oriented view, the *mapping view*, was developed. *Mapping views* can be used to build *mapping schemas*, which are capable of providing integration mappings between object models, displaying data from the generic format in specific formats, and parsing data from the specific to the generic format and for the generic to the specific formats.

The next chapter provides background information and descriptions of the formal methods and tools evaluated for use in this effort. Chapter 3 provides a detailed description of the methodology developed during this effort. The methodology is demonstrated and validated in Chapter 4 by applying it to AFRL's problem of integrating battle simulation scenario systems. Finally, Chapter 5 summarizes the research presented in this thesis, and presents some recommendations for future work.

6

II. Background

This chapter provides background information on techniques and tools available for integration of data models. The first section discusses one of the most popular commercial solutions. The next two sections deal with more general integration solutions. Finally, the last two sections discuss object-oriented and formal methods technologies, tools and techniques that can be helpful in the integration and automation process.

2.1 Data Warehouses [10]

A variety of commercial solutions to integrating similar data have become available. One of the most popular, and a good example of the commercial products, is the data warehouse concept. Data warehouses are made up of copies of information from local databases specifically structured for query and analysis. As data is brought into the warehouse, it is broken down and then built back up while grouping similar data from the different sources. As this building up is occurring, the data is "cleaned" of heterogeneity conflicts, inconsistent data, and other problems. Also saved with the data is substantial metadata which serves to help in the analysis and querying of the data. This results in data that is cleaned and ready to use without redundant or conflicting data. What a great promise!

Today 90% of the Global 2000 (the world's largest 2000 companies) are pursuing some strategy with data warehouses. This is a drastic increase from 1993, where only 5% of them were showing interest. This is largely due to the promise, mentioned above, of the technology. However, data warehouses require considerable time (2-3 years), effort, and money. Many of the companies have started their project without realizing this, and without the necessary planning and dedication. This has resulted in 85% of current database projects failing to meet their intended needs and 40% not even getting off of the ground.

In an attempt to ease the time, effort, and cost involved in data warehouses, a smaller, more focused, incremental approach has been proposed. This approach is called data marts. Data marts are more focused on specific types of information. This allows for

7

a much smaller and specialized solution and results in less time and money being required. As data marts for various areas are developed, they can be incrementally combined to form data warehouses as required. This does show much promise in being able to solve these data warehouse technologies problems.

Even with the advent of data marts, there are still some important cautions that should be kept in mind. As with most commercial solutions, data warehouses and marts are not a global solution. They do not work well for all types or purposes of integration. Data warehouses and marts are designed to take data stores of information that each individually represent a piece of the "big picture." These pieces are combined and processed to provide access to the "big picture." Data warehouses and marts facilitate querying and analyzing data but not performing data updates to the actual data warehouse or mart. All data updates are accomplished at the individual database level and are then received by the data warehouse or mart through the cleaning and combining processes. The purpose of the cleaning process is to detect incomplete, incorrect, incomprehensible, or inconsistent data. Once these problems are detected they are resolved or reported. The combining process integrates the data and may perform some summarizing. The details of both the cleaning and combining process must be defined and developed by the users. Therefore, data warehouses and marts do not simplify the integration process, but just serve as ways of approaching solutions.

2.2   Multidatabase Management Systems (MDBMSs) [8]

Over time many databases have been developed to assist organizations all over the world in conducting their everyday business. Databases exist on a variety of platforms: mainframes, workstations, PCs; different operating systems: windows based, Unix based; use various models: hierarchical, relational, object-oriented, object-relational; and are developed using a variety of commercial management systems, and even from scratch. This has resulted in similar data being stored in various databases in different ways, following different rules, having different capabilities, and using different access languages. Today's organizations are more sophisticated and have increasing data requirements and coordination. This is driving them to need to access and coordinate existing data in multiple

8

heterogeneous databases, without disrupting the operations or autonomy of the existing databases. Both multidatabases (MDBS) and Multidatabase Management Systems (MDBMSs) are being developed to meet these needs. Figure 2 shows a common structure of a MDBS.
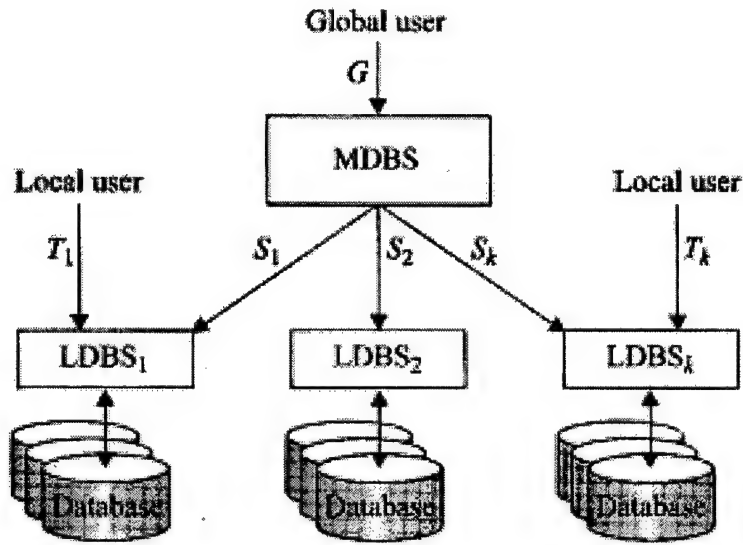
Figure 2.   A Sample Multidatabase System [8]

Past and present MDBMS approaches can be grouped into three different classes of solutions: 1) Global Schema Integration, 2) Federated Database Systems (FDBSs) and, 3) Multidatabase Language Approach. Each of these approaches must address many issues and problems, including two fundamental issues: 1) maintaining individual Local Database Management System (LDBMS) autonomy, and 2) handling heterogeneity.

2.2.1 Global Schema Integration.   Global Schema Integration was the first approach proposed. It overlooks the issue of autonomy and focuses on solving the heterogeneity solution. It involves completely integrating the component databases at the schema level to provide a single view or global schema. In this approach all of the responsibility for resolving heterogeneity lies with the Global Database Administrator and whatever team he has to help him integrate the schemas. That means that the application programmers and users get a consistent, uniform view of, and access to, the data. They are unaware that the database they are using is distributed and heterogeneous. Further, it is very easy

9

to handle data updates to the database. However the Global Database Administrator must have a detailed knowledge of the local databases' schemas and semantics. It is very difficult if not impossible to integrate the schemas fully without violating the autonomy of the local database management systems (LDBMSs). When the data is integrated, there are often semantics or characteristics of the local database that cannot be represented in the global database. This results in lost information. Finally, any time one of the local database schemas changes, or a new local database is desired to be added, the integration process must be completed all over again.

2.2.2 Federated Database Systems. Next, an approach that catered more towards autonomy and was more flexible in handling heterogeneity was proposed. This was the Federated Database Systems approach. In this approach the component database schemas are not completely integrated. Rather export schemas of the component databases are integrated according to the needs of the users. An internal command language, similar to a querying language, is added to compensate for lesser amounts of integration. Federated Database Systems are grouped into two categories, based on the amount of integration done by the Federated Database Administrator. These categories are tightly coupled and loosely coupled.

With tightly coupled FDBSs, the Federated Database Administrator has full control and responsibility for integration of and access to the LDBMSs' export schemas. When only one federated schema is developed, this approach is really just doing global schema integration on export schemas rather than the complete base schemas. There is still a uniformity of access to, and view of data. Updates can be accomplished with caution. It is also possible, in this approach, to have several different federated schemas. However, multiple federated schemas are harder to maintain and ensure the consistency of the data being sent to the LDBMSs. This approach still requires the Federated Database Administrator to have extensive knowledge of the LDBMSs' schemas (or at least export schemas), or the help of those who do, for integration. This approach makes it feasible to prevent LDBMS autonomy violations, but may still cause them during schema integration. This approach is still difficult to automate, as schema integration is still taking place. Finally,

as with global schema integration, if an LDBMS's schema changes, or it is desired to add a new one, the integration process must still be redone.

Loosely coupled FDBSs take one more step towards maintaining autonomy and dealing with dynamic schemas. With this approach it is the user's or application programmer's responsibility to create and maintain federated schemas. Creating a federated schema corresponds to creating a view against the relevant export schemas of the component databases. The federated schemas are dynamic and can be created, or disposed of, on the fly. This allows for different users to be able to have different views or semantic meanings of the data in the component databases, and to control how they are integrated. Since federated schemas can be added or dropped much more easily, it is easier to deal with changes to or additions of export schemas. In fact, adding a new export schema does not affect the existing federated schemas, unless the new information is desired by the user who developed the federated schema. However, the Federated Database Administrator's lack of control over the federated database views, and the multiple ways to access data, make it difficult to support data updates. This means the FDBMS users will be constrained to read-only access of the component databases. Also, the users and application programmers may end up duplicating much of the integration effort in each of the federated schemas. Finally, if there are a large number of export schemas, it gets confusing for users and application programers to keep track of, or find, where their desired data resides.

2.2.3 Multidatabase Languages. More recently a third approach has been developed that focuses on autonomy and leaves the problem of heterogeneity mainly to the user. This is the Multidatabase Language approach. Multidatabase language systems are more loosely coupled than loosely coupled federated database systems. Preexisting component databases are integrated without modifications. This makes integration very easy to automate and accomplish. Access to the component databases is through the multidatabase language, which is an extended query language. Multidatabase languages provide constructs that perform queries involving several databases at the same time. In fact the databases can be specified in the query. Autonomy of the local databases is maintained by default, since the only access is through queries. It is easy to deal with a change to local

11

database schemas or the addition of local databases, because the schemas are not known by the language in the first place. The disadvantages of this approach are in the responsibilities of the users and application programers. Information stored in different databases may be redundant, heterogeneous, or even inconsistent. The multidatabase language just provides query capability. It has no schema to tell it which database to look at for which information, or which information is correct when conflicts arise. Therefore, the users and application programmers must understand each database schema well enough to be able to find relevant information among the multiple databases, detect and resolve any semantic conflicts, and perform any required view integration.

2.3   Object-Oriented Technologies

As object-oriented technologies have become more developed and popular, many businesses and researchers have looked to them for a better solution to integrating heterogeneous data. This section addresses some of the object-oriented technologies, tools, and techniques that have become available and are being considered for use in integrating heterogeneous data. Also addressed are some of the obstacles encountered by object-oriented technologies and how they are being overcome.

2.3.1   UML [9, 12].   One of the main obstacles with object-oriented techniques has been the lack of standards; object-oriented modeling is no exception. Several object-oriented methods became popular, each with its own modeling notation, processes and tools. This led to many debates over which was better. This is not really an answerable question since each of the models had their own strengths and weaknesses. Usually, experienced developers took one method as their base and then augmented it with good ideas from the others. Therefore, in practice the differences between the methods were not very significant, but just served to make portability between tools, jobs and projects more difficult. Seeing this trend, several of the method gurus decided to cooperate. Eventually, the authors of the three most popular methods grouped together to build a common modeling language. These authors were Grady Booch with the Booch method [3], James Rumbaugh with the Object Modeling Technique (OMT) [32], and Ivar Jacobson with the OOSE and

12

Objectory methods [17]. One of the most impressive aspects of this effort was their ability to put aside their own methods and notations in order to achieve this standardization effort.

In January 1997, version 1.0 of the UML was released. In September of 1997, the Object Management Group (OMG) adopted UML as the notational and meta-model standard for object-oriented analysis and design. UML is based not only on the methods of the three authors (the three amigos) but also on concepts from many of the other methods. The goals of UML are:

- To model systems using object-oriented concepts.

- To establish an explicit coupling to conceptual as well as executable artifacts.

- To address the issues of scale inherent in complex, mission-critical systems.

- To create a modeling language usable by both humans and machines.
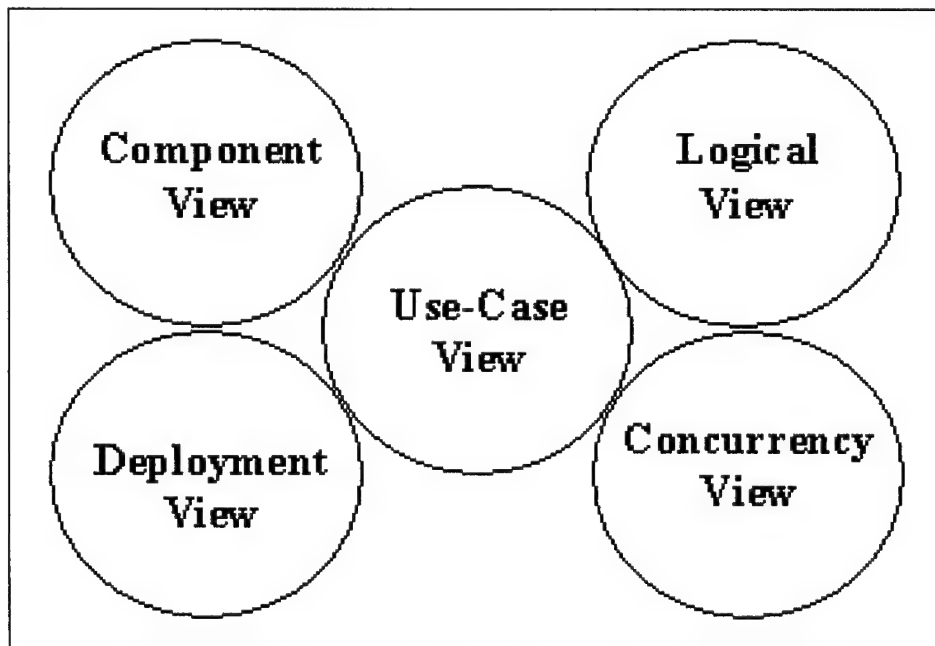
Figure 3. UML Views [9]

UML is made up of views, diagrams, model elements, and general mechanisms. The views of UML are shown in Figure 3. Each view models a certain aspect of the complete system. Each view is described using a number of diagrams. This is done because most
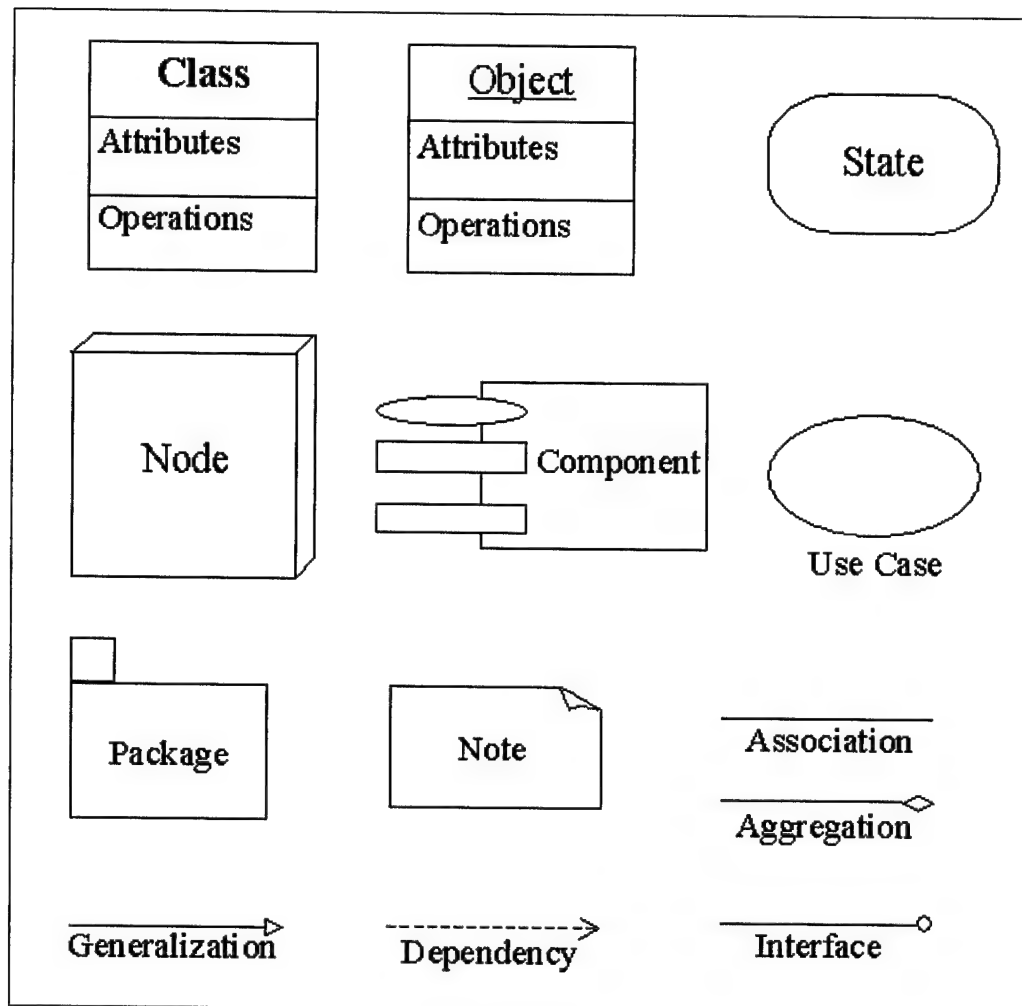
Figure 4.   UML Elements [9]

systems are too complex to capture all of the proper semantics in one graph or view of the system. Each diagram is a graph that uses model element symbols arranged to model a particular part or aspect of a system. The diagrams are part of one or more views. There are views and diagrams capable of showing structural, functional, and dynamic characteristics of a system, including interaction with external inputs and outputs. Model elements are concepts used in the diagrams with semantics that formally define them and exactly what they represent. Some example elements are shown in Figure 4. General mechanisms provide additional information which normally cannot be represented using model elements. General mechanisms include adornments, notes, and specifications.
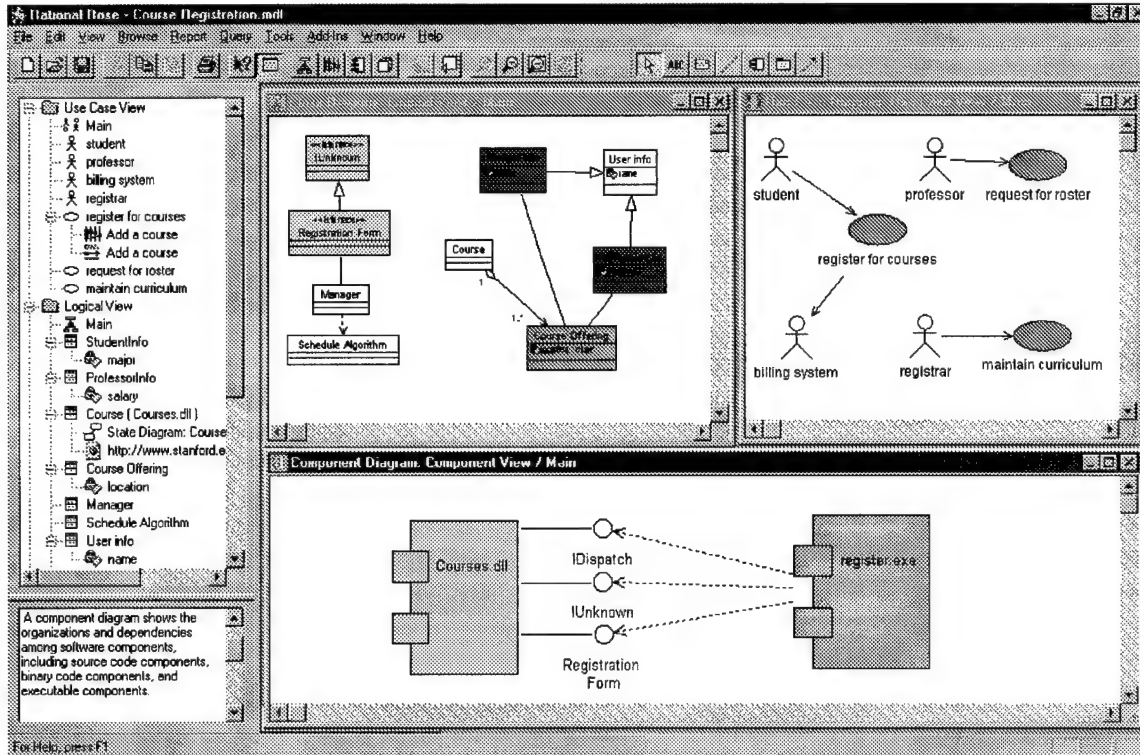
Figure 5.   Rational Rose® Version 98i [30]

2.3.2   Rational Rose.   Rational Rose (Rose) is a visual, object-oriented, and component-based system modeling tool. It is designed to use UML notation, but can use other notations such as OMT or Booch. It uses component-based design and UML to accomplish analysis, design, and implementation of object-oriented software systems. Rose has become the most popular modeling tool on the market. Not only does its object-oriented and component-based nature make it a good tool for a variety of methods and techniques, but it is also easily extended to support specific methods, development environments, and projects. Rational Rose can be extended by the following methods [29]:

- Customizing Rose menus.

- Automating manual Rose functions using Rose Scripts.

- Executing Rose functions from within other applications using the Rose Automation object.

15

- Accessing Rose classes, properties, and methods from within the software development environment by including the Rose Extensibility Type Library in the environment.

- Activating Rose Add-Ins using the Add-In Manager.

It should be noted that several of these extensibility features also support automation of the modeling effort. In addition Rose provides facilities for code generation of program skeletons. These skeletons usually consist of class declarations, including attribute and method declarations. The bodies of the methods are normally left blank, to be provided by the designer. If the model changes and the code skeleton is regenerated, it changes the declarations without losing or changing the hand-written code. This is accomplished by markers placed in the generated code identifying which parts were Rose-generated and which were human-generated. Further, using Rose's extensibility and the dynamic models, it should be possible to extend Rose to generate method bodies also. Rose code generation is currently focused on object-oriented languages, such as C++ and Java, but some SQL and Interface Definition Language (IDL) can be generated also [9]. Figure 5 shows Rational Rose 98i's user interface with several different UML diagrams being displayed.

2.4   Object-Oriented Databases [18]

Relational databases currently dominate the database market for new application development. Relational database products have been available for over a decade and a sizable industry of third-party tools have been developed. Databases are used to store data about entities. A relational database is based on tables, which represent entities. The rows of the tables represent instances of the entity and the columns represent the attributes of the entity. Each column has a data type called the column's domain. Unfortunately, application programming languages are not based on tables and in fact are moving towards object-oriented technologies. In object-oriented languages, entities are represented by classes, with objects (instances of the class) representing instances of the entities. Each class has attributes and methods. The attributes represent the attributes of the entity and the methods represent the actions that the entity can take.

16

Having different data models for the database and the application program leads to complications, and requires a mapping to be accomplished between the two models. From the relational database's perspective, there is only one model, one fixed type system, and one language, all of which it dictates. Therefore, all mapping must be handled by the application programs. This drives the application developer to make a lot of additional decisions and to implement a lot of additional software. As a result, the application developers and end users who wish to interact with the database using SQL must have a good understanding of both models and the mapping between them.

Another problem is found with the limitations of the relational model to handle complex objects and operations. Even after the data has been mapped from the application to the relational tables, the operations that can be performed on these tables, and the data types that can be used for the data, are restricted by the relational database and SQL. Therefore, much of the the flexibility of the programming language must be lost by using easily mapped types and operations, or data manipulation must be done almost entirely in the application with complicated mappings back to the relational database.

The relational community has been working on an SQL3 standard that expands the relational model to use objects, and many of the vendors have provided their own extensions to support objects. The goal is to use the benefits of object-oriented techniques without loosing the heritage, maturity, and capabilities of proven relational tools and techniques. Unfortunately, the extensions generally just allow objects to be used as domains, with the instances still saved in columns. This is still very restrictive when compared to the object-oriented paradigm. Also, objects created by using object-relational extensions are still not directly compatible with the objects found in object-oriented programming languages. Therefore, two models and a mapping must still be maintained.

In an attempt to correct these problems, object-oriented databases have been developed. These databases are based on database transparency. Database transparency allows the application program and database to use the same programming language and object model. This removes the need for the second object model and the mapping between them, which results in substantial savings of development cost, time, and effort. It also allows the full flexibility of the programming language to be used in both the application program

17

and database functions, including defining new classes to be used as types for attributes of other classes.

In theory, the object-oriented databases solve all of the problems relational databases run into in dealing with complex objects and mapping to application languages, without losing any existing capabilities. In practice, object-oriented databases still have some problems being fully compatible with object-oriented languages, without any mappings. This is due to the way in which objects are made persistent. Usually, objects must inherit from a persistent class provided by the object-oriented database management system (OODBMS) and not available in the standard programming language. Also, the use of some types, such as pointers, cause problems with persistence and must be replaced by OODBMS provided types. Finally, object-oriented databases lack the standardization found in relational databases. Therefore, not only are the object-oriented databases not fully compatible with the application languages, they are not compatible with each other.

It is very controversial whether object-oriented databases have succeeded in maintaining all of the relational database capabilities. Many people question the abilities of OODBMSs to handle querying, integrity constraints, and views. The answer to these issues, and the lack of standardization, lies in the fact that OODBMSs are very new when compared to relational systems. When relational systems were just getting started, they faced many of the same problems. Recently, efforts have been made to standardize the OODBMSs. The Object Database Management Group (ODMG) was formed in the summer of 1991 to improve progress towards a standard. Their first release of an object-oriented database standard was the ODMG-93 version 1.0 in 1993, which was adopted by the Object Management Group (OMG) as the standard interface for storing persistent state in February 1994. Since then versions 1.1, 1.2, and 2.0 have been released. A complete reference on ODMG 2.0 is found in [5]. Part of the ODMG standard is the Object Query Language (OQL) which was adopted by the OMG in May 1995.

Along with the standardization effort, a great deal of research and development is also being done for object-oriented databases. One main focus area has been in defining and using views. Many techniques have been proposed for defining and using views. Of particular interest to this effort are the concepts for using views presented by Giovanna

Guerrini, Elisa Bertino, and associates in "A Formal Model of Views for Object-Oriented Database Systems" [15], which is based on previous work of Elisa Bertino [2]. They propose 3 classes of views:

Object-Preserving Views: Classes whose attributes are extracted from existing objects.

Object-Generating Views: Classes whose attributes create new objects.

Set-Tuple Views: Classes whose attributes define a relationship between existing objects.

Each of these classes of views are based on views being created as object-oriented classes and being placed in their own inheritance hierarchy, separate from the base class hierarchy. This prevents the confusion of where to put views in the base hierarchy and prevents the base hierarchy from becoming unnecessarily large due to many intermediate classes that add no semantic meaning.

They further propose combining these views into aggregation hierarchies to create schema views, also known as virtual schemas. These views and virtual schemas can be used for shorthand queries, in the integration of heterogeneous databases, to dynamically modify the database schema while maintaining its older versions (dealing with schema evolution), and in creating external or export schemas. This provides the capabilities offered by traditional relational database views with the addition of integrating heterogeneous databases.

2.5 Formal Methods

Formal methods are mathematically based techniques for describing properties of systems during development. They are used to specify, design, verify, and maintain systems. There are basically two ways to apply formal methods to the design process. The first is to formally specify the system. The formal specification is then used as the basis to design the system. Once the system is designed, formal verification is used to prove that the designed system satisfies the formal specification. The second method is to again begin by formally specifying the system. Then the formal specification is used as the starting point for formal development (synthesis). This means that the specification undergoes a

series of transformations to executable code. Each transformation is based on rules that are verified to preserve semantics. Therefore, the executable code is correct by construction and formal verification is not needed. Since formal verification can be very costly in time, effort, and money, transformation systems have been the focus of the research efforts at AFIT [14].
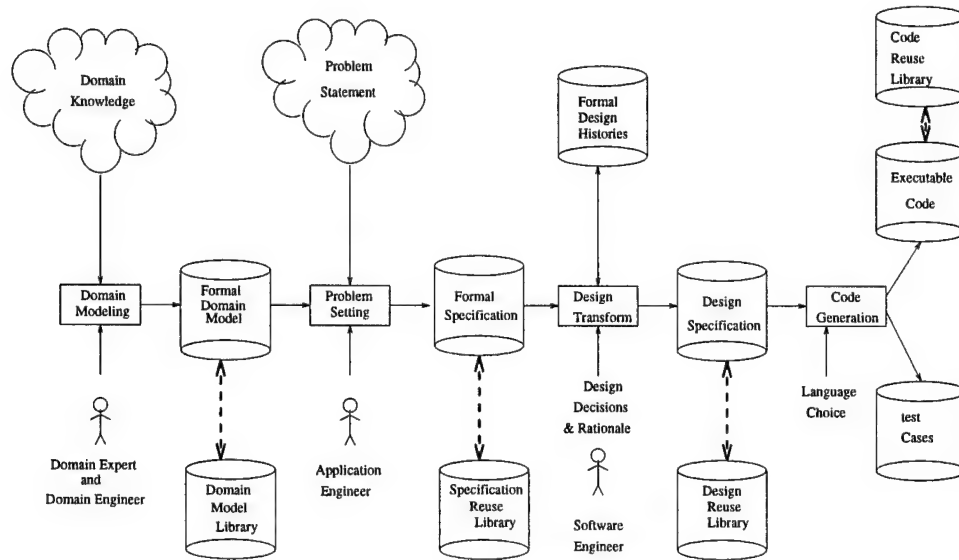
Figure 6.   Typical Transformation System [16]

2.5.1   Formal Transformation Systems [16].   Figure 6 shows what a typical transformation system would involve. The system shown actually starts before the formal specification, with domain models. This allows for reuse of specification elements in different specifications based on the same domain. In fact, each step of the transformation system makes use of as much reuse as possible and maintains a reuse library. Another goal of a formal transformation system is to automate as much of the work as possible. Much of the lack of wide spread acceptance of formal methods is due to the intense effort involved in using them. The tedious detail of formal methods makes them error prone when used by people. Automation can take care of the tedious details and reduce the effort of determining design decisions. It is possible to automate some design decisions, or at least to assist with the decisions, but to date it has proven unreasonable, if not impossible, to automate all design decisions.

20
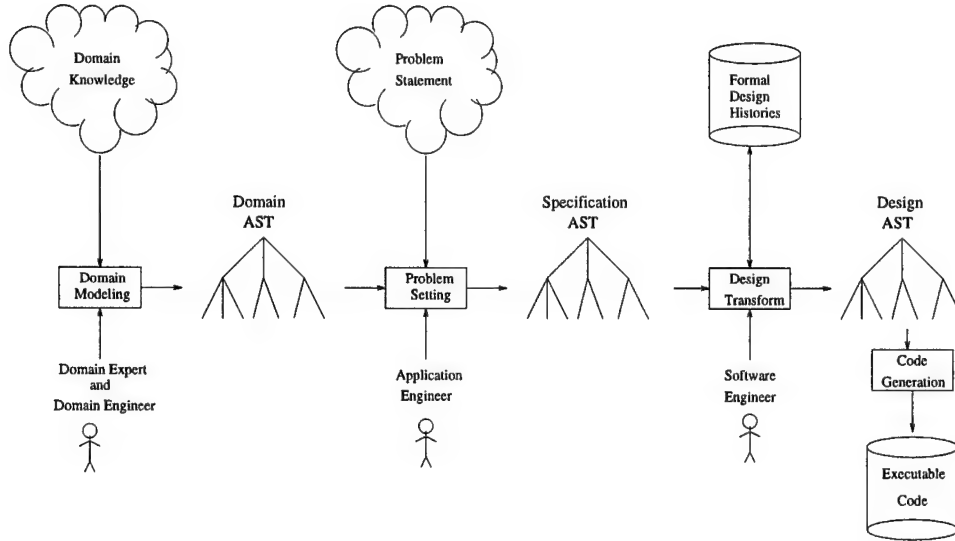
Figure 7.   AFIT's AST-Based Transformation System (AFITtool) [16]

2.5.2   AFITtool [1, 16, 24].   AFITtool, shown in Figure 7, is a formal transformation system, based on abstract syntax trees (ASTs), which has been developed at AFIT. It actually has components for forward and reverse engineering, but only the forward engineering components are shown and discussed. AFITtool was built using Software Refinery, particularly the REFINE™ language [31]. First specifications must be accomplished in LATEX Z (pronounced Zed) format. LATEX is a mark-up language developed by Leslie Lamport in the mid 1980's and is based on TEX. TEX is a powerful text formatting program developed by Donald E. Knuth [21]. Z is a formal specification language developed in England and based on predicate logic. Some work has also been done which allows models developed in Rational Rose to be exported into LATEX Z files [24:57-61].

Once a LATEX Z specification is obtained, the specification model is parsed into AFITtool by creating a sub-AST for each class of the model and storing it in the domain AST. The domain AST represents the domain being modeled in the system. Once a domain model has been developed and parsed into the domain AST, the Elicitor Harvestor subsystem may be used to interactively refine a specification from the domain based on a problem statement. This refined domain AST is considered the specification AST.

Using a series of correctness-preserving transformations, the specification AST is transformed into the design AST. Once a design is in the design tree, design transforma-

21

tions and code generators are used to generate executable code in a variety of programming languages. To date, programs using subsets of the Ada and C++ programming languages have been generated.

Currently, an effort has begun to develop a Java version of AFITtool. This new version will have one wide-spectrum AST which contains the elements needed for the domain, specification, and design trees. The wide-spectrum ASTs will be stored in a repository that is based on the Java API of ObjectStore, an OODBMS (see Section 2.4). It will also be possible to store the ASTs in files using persistent Java capabilities. This new version of AFITtool is referred to as AFIT's Wide Spectrum Object Modeling Environment (AW-SOME). By being Java-based, AWSOME will avoid requiring users to maintain expensive Software Refinery licenses and will allow for development of more user-friendly, powerful interfaces using Java's GUI development packages.

AWSOME will be able to parse in specifications written in Z and possibly OQL. AWSOME will also be able to parse in and generate programs written in the Common Object-Oriented Imperative Language (COIL) [13]. The COIL was developed at AFIT to serve as an intermediate language from which other programming languages could easily be generated. The AWSOME wide-spectrum AST is based on a COIL AST, expanded to handle domain specification elements. A COIL parser is currently being developed, but was not completed in time to use with this effort.

2.6 Summary

This chapter provided background information on several tools, techniques, and technologies currently being used, or potentially able to be used, to integrate similar data in heterogeneous formats. The next chapter evaluates these tools, techniques, and technologies and how they can be used to integrate similar data for the purposes of translation between heterogeneous formats. It then presents a methodology based on these tools techniques and technologies, as well as the previous methodologies of past AFIT efforts presented in Chapter 1.

III. Integration Methodology

Although a general solution to integrating similar data with heterogeneous models has been explored for 20 years [8:145] or more, none has been found. However, several solutions have been developed for specific cases and some for solving classes of problems. The solutions for classes of problems tend to involve methodologies and techniques requiring great amounts of knowledge and time spent writing code. The expense in time, effort, and knowledge for general solutions to integration has resulted in most efforts failing, or just integrating specific cases (not as costly for each case, but very repetitive and costly to do many times). For this effort it is assumed that it is still unreasonably difficult to develop a methodology that is optimal for all integration situations. However, it is reasonable to develop methodologies that are good for a general class of integration problems. One such class of problems occurs when data exists in multiple formats, each being used by tools based on its specific format. The tools are only able to use data in their base format, but it is desired to use data from all the formats in each tool. This class of problems is called the "data format translation problem" in this work.

Section 2.2 discussed the variety of databases developed over time, the various heterogeneous platforms and formats they exist on, and the need to merge or collaborate the data between them. This is a subclass of the data format translation problem. However, databases are just one form of data storage and all the various forms of data stores have this need. Many of these data stores would be built with common formats, or even as a single data store, if they were being developed today. However, the loss of legacy data and tools, and the effort to convert data and tools to use a new format, are both unacceptable. Providing a method to translate data from one format to another with minimal time, effort, and cost, would solve this problem. Data could be translated from one system to another, allowing the data stores and the tools using them to take advantage of other data stores and their information.

This chapter presents a tool-based methodology for solving the data format translation problem, by integrating the various data formats into a combined format. The methodology also provides for automated code generation of methods that parse local

23

data into the global format, provide local views of the global data, and export global data into the local formats. Data format translation problems have the following qualities:

- There exist multiple data stores with similar data, but different formats. These data stores are called *local data stores* in this effort. Similar data means that at least a subset of data in each data store is also data found within the other data stores.

- The data stores may be databases, but usually are a variety of different formats.

- It is desired to use existing data from the various data stores with tools based on specific data store formats.

- It is desired to enter future data once into one data store and have it available for tools based on each of the local formats. This data store is called the *global data store*.

- Translation of data, not operations on the data, from one format to another is desired.

- There is no need for the data in the global data store to be a current reflection of the data in the local data stores. In other words, there is no need to automatically update the global data store due to local data changes or local data stores due to global data changes.

- It may be desirable to manipulate data in the global format, or develop tools that use data in the global format.

The chapter begins with a discussion of the tools and techniques on which the methodology is based, and why they were chosen. The methodology is then presented, including the main integration participants (an integration tool being one of them) and their roles. The chapter ends with a classification of and resolutions of heterogeneity conflicts.

3.1 Background

3.1.1 Using a Global Data Store. Colonese's [6] and Weber's [38] theses were based on integrating sensor-based engagement-level simulation scenario data formats. However, the goal was to allow data to be translated from one simulation format to another. If

the goal is translation, why did they bother integrating the formats and developing global data stores? Why not just develop a translator? These were follow-on theses to Park's thesis [26]. Park did not do research on integrating battle simulations, but on translating scenarios between battle simulations, and in his thesis is the answer to why this would lead to follow-on theses based on integration. For his methodology, Park decided to translate scenarios into a middle generic format first, and then from the generic format into other simulation formats. This resulted in his research performing integration to develop the middle generic format.

Park determined that this two-step translation process was better for two main reasons. First, it is unreasonable to try and develop a generalized translator due to the specialized and heterogeneous nature of each simulation's scenario data structure. He determined there were two ways to reasonably handle this problem. Either write a translator system which has separate translators for going from each simulation system to each other one, or develop a generic middle store to which each simulation format is translated before being translated to another simulation format. The first method results in a total of n^2 translators being needed. Also, each time a new simulation system is added to the existing n systems, $2n$ new translators will be needed. With the generic format, only $2n$ total translators are needed with only 2 new ones required for each new simulation. Although developing a middle format is more complicated than performing one-to-one translations, if more than a few formats are to be integrated, an over all savings of time, effort, and cost is achieved.

Further, Weber discovered that individual scenarios do not always contain all the data needed for another scenario. In fact, the new scenario created from the translation of the data in the other format is only useful under one of the following circumstances:

1. The two formats contain the same data, just in different formats.

2. The target format's data is a subset of the source format's data. (Translation will only work one way.)

3. The target scenario can be used with only partial data, and the source scenario provides all required data.

4. There is some way to add additional needed information to the source scenario.

Having a middle (global) format in which data can be manipulated provides a means to add additional information to the data imported from one scenario format. This allows Item 4 to always be met regardless of the rest. Further, this allows for new data to be gathered in the global format and then translated to any format desired. Also, any new tools developed could be developed to use the global format. This prevents the legacy data and tool problem from growing larger. Finally, a new data format and tools could be developed to replace the old format and tools, without the hindrance of backward compatibility. The new format could just be integrated into the global format. Then all legacy data could be translated to the new combined system.

3.1.2 Using the Global Schema Integration Method. Providing a global data store lends itself to using the global schema integration method presented in Section 2.2 for integrating the models. However, this was described as being the poorest method of the three multidatabase techniques described. That was based on its failure to preserve local autonomy, handle local schema evolution, and facilitate automation. Federated databases and multidatabase languages handle these problems better, but do so at the cost of additional knowledge requirements of users and application programers, and make it much harder to present a uniform view and access to local data. These disadvantages are not desired either. Further, it is not always possible, or at least easy, to allow manipulation of local data at the global level using federated databases or multidatabase languages. As described above, manipulating data at the global level is very important to the data format translation problems.

Looking at the description above of the data format translation problem, it is seen that concurrent access to data by local and global data stores is not needed. Further, to translate data from one format to another requires the schema integration administrators to have a detailed knowledge of the local databases regardless of which integration method is used. Therefore the autonomy issue is a much more minor issue for the data format translation class of problems.

The purpose of integrating the formats is for translating data between various formats. If the local data store schemas are very dynamic, then the changes occurring to the schemas can be directed to bring the formats closer together. This will lead to the individual formats eventually developing into a common format. If the schemas are constantly changing, then tools are either constantly changing, or are not heavily dependent on the local data schemas. If the tools are constantly changing, the changes can be directed to eventually allow the tool to work with a common format. If the tools are independent of the local data schemas, they should be able to use data in the other formats. Therefore, to have a data format schema translation problem would imply that local schemas are fairly static. However, it does not mean they never change. In addition, the problem of local schema evolution also includes how to handle integrating an additional schema after the integration of the initial set of schemas has taken place. This is a definite problem and is discussed in the following sections.

3.1.3   Using a Transformation System.   Commercial systems and models are available to assist the automation of integrating and processing data. The most established of these, and a good representation of the rest, is data warehousing and data marts as described in Section 2.1. However, Section 2.1 also describes them as time-, effort-, and cost-intensive. The incremental concept of data marts is a desired quality. The transformation tools discussed in Section 2.5.1 provide for incremental ability also. Transformation systems can also take a great amount of time, effort, and cost to develop, but can be designed for generic use. For example, AFITtool described in Section 2.5.2 is a generic transformation system able to be used with data schemas or many other types of designs and problems. Therefore, if AFITtool or another existing transformation tool were used, it would only require the addition of integration and heterogeneity conflict resolution abilities. These would include user input interfaces and the ability to process the user's input. Information could then be provided to the tool directing how the integration should take place. This directing involves identifying what classes and attributes should be integrated together, which is the part of global schema integration that is very difficult, if not currently impossible to automate. However, once this is done, the transformation

27

system would be able to generate the resulting schema code automatically in a variety of programming languages and formats.

This schema generation capability maximizes the automation and minimizes the human involvement in integration, which in turn minimizes the automation problem of the global schema integration method. Further, the automation of code generation provides a way to deal with schema evolution. When there is a new data store schema to be integrated, the integration can be accomplished and then the code regenerated by the system. When existing schemas evolve, they could also be re-integrated, resulting in data from both the old and new schemas being able to be parsed into and from the global data store.

There is still a problem. If the new integrations drive changes to the global schema, then all of the existing integrated schemas must be re-integrated. To solve this problem, the transformation-based tool must have the ability to recognize how global schema changes impact the existing integration of schemas, and the ability to correct the integrations accordingly. These abilities are facilitated through the use of mapping schemas, described in the next section.

3.1.4 Using Mapping Schemas. Section 2.4 discussed a method for developing views for object-oriented data stores. Object-preserving, object-generating, and set-tuple views were presented.

Specifically, for this effort a new object-oriented view, the *mapping view*, is proposed. The mapping view is basically an object-preserving, set-tuple view. It is an exact replica of a local schema class, except that its only attributes consist of one local object attribute that contains an instance of the equivalent local class, and one or more global object attributes that contain the global classes with one or more attributes mapped to attributes of the local class. The only other attributes optionally allowed are those needed for displaying purposes. These additional attributes violate the object-preserving rule of coming from existing classes, but since they are for display purposes only and contain no new information, the object-preserving concept is still maintained. The methods of a mapping schema consist of five sets:

1. Regular *gets* and *sets* for the local and global object attributes.

28

2. *Gets* and *sets*, corresponding to the original local attributes, that call the *gets* and *sets* of the global attributes mapped to the local attributes.

3. *Import* and *export* methods that call the *gets* and *sets*, respectively, of the corresponding local attributes.

4. *Initialize, import,* and *export* methods to handle building the mapping schema and the transfer of data between the global and local schemas.

5. (Optional) Methods used to handle the graphical displaying of the mapping view.

An example of a mapping schema is shown in Figure 8, and is explained in Section 3.2, Step 3. The use of mapping views and schemas allows integration to be completed without changing the local schema. Any needed changes should be made unnecessary by using the mapping schema, or can be made to the mapping or global schema instead of the local schema. This protects the autonomy of the local data stores against schema changes, which is really the only form of autonomy that needs to be preserved.
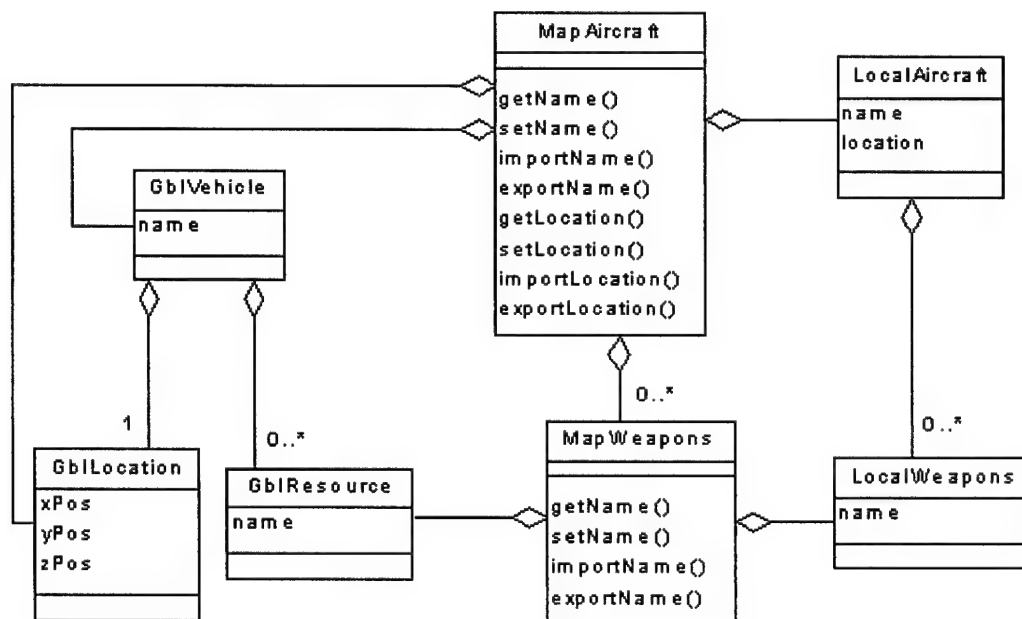
Figure 8.   Example of a Mapping View

Finally, the mapping schemas contain the integration information from local schemas to the global schema. When a new integration causes a change to the global schema, the

29

transformation-based tool can use this information to detect which mapping schemas were affected by the change, and update the mapping schema accordingly.

3.1.5 Using an Object-Oriented Common Format.   The mapping views and schemas above are based on the local and global schemas being object-oriented. It might be questioned whether requiring the local schemas to be translated into a common form before integration is efficient or just adds extra steps. Integration of heterogeneous schemas is difficult enough without having to worry about format differences as well. Dividing the integration into these two parts greatly reduces the complexity of both of the processes. This is a common practice and is used by many methodologies [6, 8, 38:122].

A similar technique could be used with relational databases using relational views, but most other data store formats do not have a view capability. There are four main advantages to using a object-oriented common format:

1. The use of UML and Rational Rose has become very popular in commercial industry.

   Companies have begun developing tools which are based on UML or interface with Rational Rose. Also, Rational Rose's extension and code generating capabilities, discussed in Section 2.3.2, make it very flexible to use and modify to meet the user's needs. In fact, much of the functionality of a transformation system can be built into Rational Rose. Although UML and Rational Rose can be used with relational databases, and have a few tools to help with that, they are mainly based on and used with object-oriented methods. This provides an ability to develop models for the integration tool that are compatible for use with many other tools and processes being developed by commercial companies.

2. OODBMSs and object-oriented programming languages use the same data model.

   One of the objectives of automating the integration effort is to allow the global data store, with its views and parsers, to be developed in many different languages and formats. This allows the global data store to be used on many different hardware and software platforms. Often tools are developed for use in conjunction with commercial products. This allows for greater capabilities but for a lower cost and effort. However,

30

commercial tools often become unavailable or increasingly expensive. The automated generation of the global data store and tools allows for them to be used in conjunction with a commercial tool while beneficial, and regenerated in a format independent of the commercial tool when it becomes unavailable, out of date, or too expensive. By OODBMSs and object-oriented programming languages using the same (or at least very similar) data models, the global model schema can be generated to be used by OODBMSs or object-oriented programming. This allows OODBMS capabilities to be used with the global data store, but maintains the ability to use the global data store without the OODBMS.

3. Object-oriented models group operations (methods) with the data (attributes).

   The ability to generate methods to go with the data attributes provide many benefits. The integration tool uses methods to handle many of the heterogeneity conflicts between schemas. The integration tool also uses methods to generate views and parsers between the local and global data stores. Further, because the methods are associated with the proper data, after integration the local and global data store users do not have to know the details or even the existence of the methods. They only know they receive the proper data when requested. This same principle is a big part of why object-oriented technologies are better able to handle large complex data than relational databases.

4. AFITtool is based on using object-oriented models.

   Although there is no requirement to use AFITtool for this methodology, it is a readily available transformation system that can be used for a demonstration of this methodology with full control of the code and no additional training required.

   3.1.6 Basis of Methodology.   In accordance with the above discussions, this methodology is based on using global schema integration of object-oriented local schemas into an object-oriented global schema. This is done through the use of mapping views and schemas and a transformation-based integration tool. The methodology assumes the existence of object-oriented models of all local data schemas to be integrated. Translation of local schemas to local object-oriented schemas, and parsing data from local data

stores to local object-oriented data stores are considered out of scope for this effort. This methodology is designed to be used to solve data format translation problems by providing a global format into which data can be entered, parsed from local formats, or modified and then parsed out to any integrated local format.

The mapping schemas and transformation system that are the basis for this methodology were developed for use with formal methods. By requiring an interactive integration tool to be used with the methodology, and focusing on the integration of schemas only and not data, the formal nature of the transformation system and mapping schemas may be bypassed. This allows for this methodology to be executed without requiring the integrators to have formal methods training. However, there are some additional benefits to extending the methodology to use formal methods. These benefits are discussed in Chapter 5.

3.2   Methodology

For this methodology, integration of data stores is accomplished through the combined effort of an integration tool and a team of local schema experts. Figure 9 shows the general process which the methodology defines. The dotted lines in Figure 9 represent areas not defined or performed by this methodology. These include the following:

- Developing the object-oriented schemas for the non-object-oriented local data stores (the format path).

- Developing data parsers to transfer local data between the non-object-oriented and object-oriented formats (the data path).

- The development of end-tools and determining the required input of end-users.

Also, although parsers are provided to transfer data between local and global formats, no ability to integrate different data instances is developed. In other words, this methodology performs schema integration and considers data integration out of scope.

The major participants and the steps of the methodology are described below:

Schema Integration Administration Team (SIAT)

The SIAT consists of at least one expert for each local schema to be integrated,

FORMAT PATH
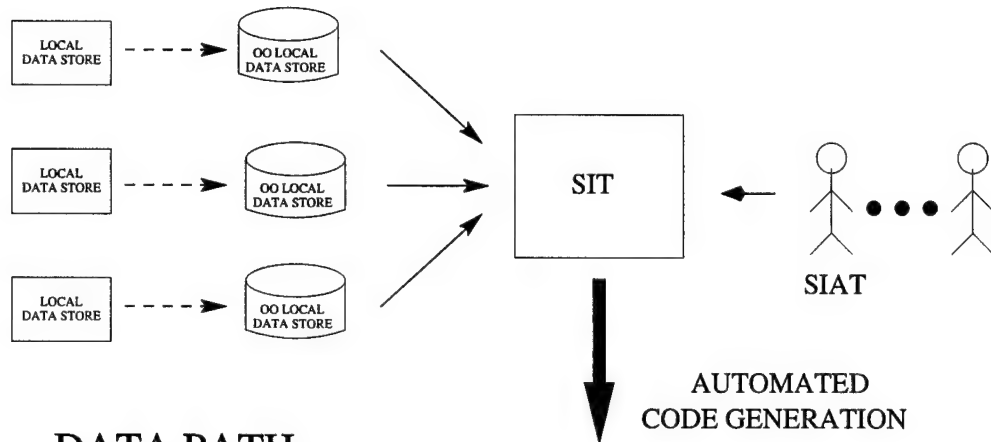
DATA PATH
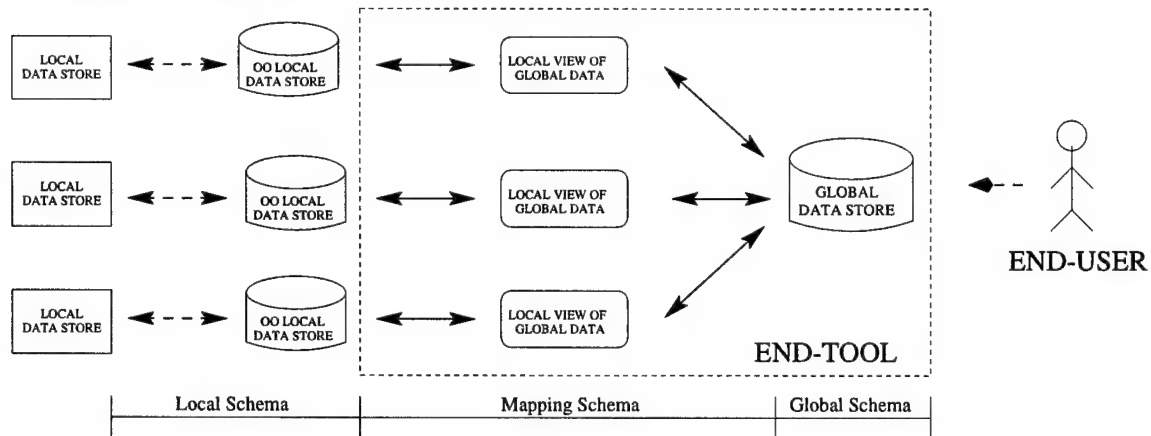
Figure 9.   The Tool-Based Data Store Integration Process

a chairman, and assistants. The chairman and assistants should be experts in the global schema. The assistants provide global schema expertise during integration (without the assistants, the chairman would have to participate in every integration to provide global expertise). The SIAT is responsible for the following:

1. Determining the goals of the integration.

2. Selecting which local data stores will be integrated.

3. Developing object-oriented schemas and data stores for non-object-oriented local data stores.

33

4. Using the integration tool, described below, to integrate the object-oriented local data stores.

Schema Integration Tool (SIT)

The SIT is a transformation-based tool. As shown back in Figure 9, the SIT should have the ability to accept local models as inputs, accept mapping and heterogeneity resolution information from the SIAT members, and output a global domain. The global domains consist of a global schema, one or more local schemas with a mapping schema between each local schema and the global schema, view generators, and data parsers. The global schema defines the global data store. The mapping schemas define local views of the global data. The view generators are methods of the mapping schemas, which build a mapping schema view over an instance of the global schema data. The data parsers are also methods of the global schemas. They are used to parse local data into the global format and export global data into local formats. The local schemas are generated exactly as provided to the SIT. The SIT is not allowed to modify local schemas. The local schemas may already be in use and have tools developed to use them. If parser methods are provided with the local schema to transfer data between the non-object-oriented and object-oriented local formats, code is generated for them also. However, the development of these parsers is not part of this effort. The SIT should automate as much of the integration effort as possible, including all method, mapping schema, and code generation. The SIT should have the ability to save completed global domains as well as global domains with partially integrated local schemas. The design of the SIT should be kept as simple as possible to minimize time, cost and effort. If more capabilities are found to be needed, they can be incrementally added or worked around for a one-time need. If additional capabilities are added, they should not change the process used by the SIAT members to integrate local schemas other than allowing previously restricted actions, and asking the SIAT members to provide answers to simple questions. The questions serve to provide the SIT with the knowledge the SIAT member used to identify the need for the requested action.

34

Schema Integration Tool Administrator/Developer(SITAD)

    The SITAD is responsible for developing, maintaining, upgrading and administering the SIT. This is the person who has to have a good knowledge of formal methods, transformation systems, programming languages, integration, etc. Each time a change is required to the SIT, the SITAD must obtain a good understanding of the change and how to accomplish it. This may require training, research, or the help of the SIAT. A lot of time and effort are spent by the SITAD, but once the SIT is operational, time is saved on every integration. The SITAD may have a team of specific language programming experts and other specialists. The key is for any change needed to the SIT to create work only for the SITAD, while decreasing the work and effort, or increasing the abilities, of the SIAT members during future integrations.

End-Tools and End-Users

    End-tools are tools that operate on the local or global data stores. They are not involved in the data format integration, but will use the integration and code that has been generated to translate data between local stores, or input data in the global format that can be used by any of the local stores. As shown back in Figure 9, the code generated by the SIT serves as the "guts" of the end-tools. End-users are the people using the end-tools.

Preparation Step 1: Goal Determination

    Schemas from at least two data stores should need to be integrated, with the possibility of more schemas being integrated later. Before any integration is done, a reason and goal for doing so should be identified. The general goal is to allow data in one format to be used by tools based on another format. However, there should be more detail to the goal. Rarely do different data stores contain the exact same data in different formats. On the rare occasion in which this is the case, the goal would be to fully integrate the models and expect all needed data for each model to be available from every other model. In the more usual case, this is not true. Often just a subset of the data is desired to be used with the other data stores. In this case it must be determined what information is to be integrated and what semantics it must carry

with it to make it possible to identify where it would fit in the other model. The demonstration of the methodology in Chapter 4 discusses example implications of not doing this. The SITAD and SIAT should both be a part of this step, with the SIAT chairman making the final decisions.

Preparation Step 2: Develop Models

Once the purposes and goals for integrating the local schemas have been determined, the object-oriented models of the data stores' schemas should be developed. It is important that the purpose and goals in Preparation Step 1, not just the data store structure, be a driving factor in how the models are developed. Doing so can greatly reduce the time, effort, and cost of the integration process. This is discussed more in Chapter 4. The members of the SIAT should each handle their own data store, but coordinate with other SIAT members to make the models as semantically rich in the integration areas as possible. Research has been accomplished [38], and is underway [23,27], to develop methodologies that can assist in preparing these models.

A few items should be noted. First, some data stores may already be based on an object-oriented model. The preparation steps should still be performed with such a data store. If it is one of the initial data stores, it still provides valuable information for determining goals during Preparation Step 1. If it is one of the later data stores, it is important to ensure that it makes sense to integrate it. Also, it may be desired to integrate only part of the model from the data store. Creating an export schema for just the desired portion of the model is recommended. This presents an opportunity to ensure the export schema provides the proper semantics needed to identify relations to the global model.

Second, these two preparation steps should be repeated each time a new local schema is to be integrated into the global schema. This ensures consistency of the goals of the integration and the semantic levels of the local schemas. Consistent goals and comparable semantic levels reduce the integration time, effort, and cost, and maintain the usefulness of the global model.

Further consideration of Preparation Steps 1 and 2 are out of scope for this effort and are assumed to have already been accomplished before integration starts.

Step 1: Prepare the Schema Integration Tool (SIT)

One major focus of the methodology is to put most of the effort, cost, complexity, and variance into the development of the SIT. This allows for the human involvement and required training to remain the same. The result is expending increased time, cost, and effort on the tool once, and saving time, effort, and cost during each integration. This results in increasing savings as the number of integrations for which the SIT is used increases. Still, the SIT should be developed incrementally, implementing only those capabilities which are frequently required or may be added without increasing the complexity of the SIT. This reduces cost and complexity of developing and maintaining the tool. It is very easy to get trapped developing complicated and costly capabilities into the SIT that will probably be used only once or twice, if ever. With the many possible conflicts and resolutions that exist, this could quickly drive the time and cost of the effort so high as to not be worth the benefits of the integration.

If a SIT is not already available, develop one with the capabilities required to do the integration of the current models. Only those capabilities currently needed or that can be easily added should be developed. If a SIT is already available, determine if any additional capabilities are needed and add them. As integration proceeds, the SIT may be revisited and have additional capabilities added. The addition of these capabilities should not add additional work for the SIAT members. They should only provide the SIAT with additional capabilities or request the information on which the SIAT member based their actions.

Select the most broadly based local schema, enter it into the SIT Tool and convert it to the global model. The SIT should have the ability to perform this conversion automatically, as well as to provide the trivial mapping between the local schema and the identical global schema. Additional SIT requirements are discussed in the other methodology steps and in the heterogeneity conflict resolutions section.

Step 2: Compare, Prepare, and Resolve

To start step 2, another local schema must be loaded into the SIT and converted into a mapping schema. This conversion should be handled automatically by the SIT.

The mapping schema will appear identical to the local schema except for being called a mapping schema. The conversion allows the SIT to create a mapping schema shell, link it to the local schema, and prepare it for integration with the global schema.

Once the SIT prep work is completed, step 2 can continue. First the SIAT member(s) compares the global and mapping schemas to identify inter-schema relations and conflicts in the related data's representation. This requires the SIAT members' expert knowledge of the local and global data syntax and semantics. Since the local schema experts may not be fully familiar with the global schema, the combined effort of local and global experts will be required.

The global schema is then prepared for integration. This is accomplished by selecting resolutions for all the heterogeneity conflicts that have been discovered, and making any changes to the global schema called for by those resolutions. Figure 12, pg. 43, shows a classification of heterogeneity conflicts, which is described in Section 3.3. When resolving conflicts, all schema conflicts should be resolved first, then class definition conflicts, then attribute domain definition conflicts. Resolutions are not needed for naming conflicts since the attribute mapping discussed in the next section solves them automatically. By resolving the conflicts in this order, the more complicated conflicts are resolved first. This is important, because the more complicated conflicts are often made up of and associated with multiple less complicated conflicts. In the process of resolving the complicated conflicts, many less complicated conflicts are often resolved and/or created. Specific heterogeneity conflict resolutions are discussed in Section 3.3.

Step 3: Map

Once Step 2 has been completed, the schemas should be prepared for integration. This is accomplished through linking and mapping. Figure 10 shows a simple example of a mapping view. Three global schema classes are shown. *GblVehicle* is an aggregate class with a basic *name* attribute, a component set of *GblResource* classes, and a component *GblLocation* class. The *GblLocation* class represents the x,y, and z coordinates of the *GblVehicle* with respect to its assigned base. A local schema is being integrated to the global schema. The local schema consists of a *LocalAircraft*

38
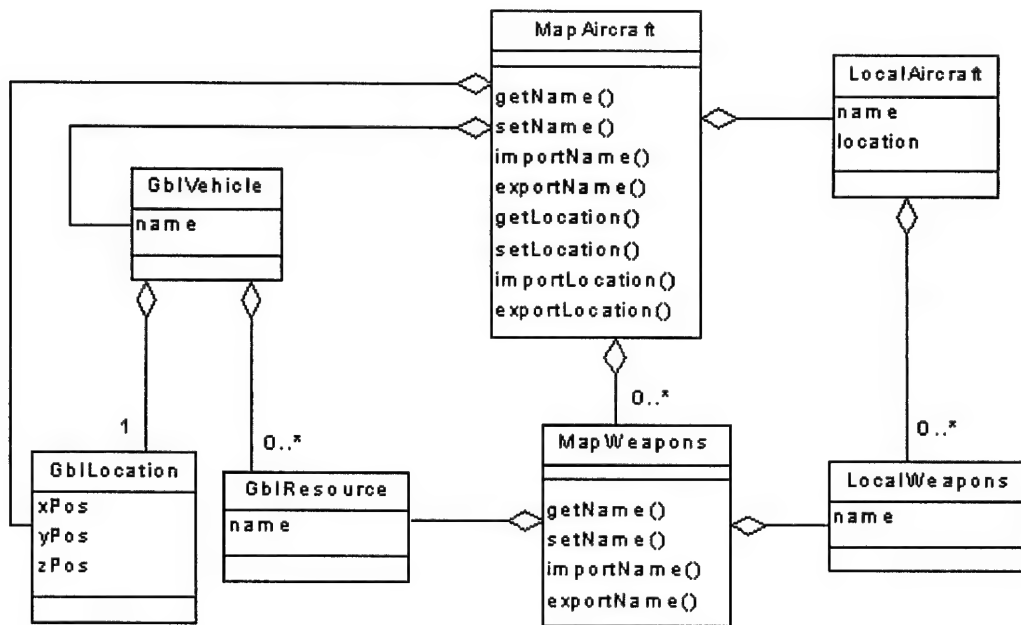
Figure 10.   Example of a Mapping View

class with basic *name* and *location* attributes, and a component set of *LocalWeapons*. A mapping view is needed to map the *LocalAircraft* class to the *GblVehicle* class. The mapping view should have the same aggregate structure as the local schema. Instead of having basic attributes, the mapping schema should have a reference attribute to the local class and the global classes with the corresponding data. The mapping view also has *get* and *set* methods, which make it appear to have the same attributes as the corresponding local class. These *get* and *set* methods actually access global attributes that contain the corresponding data to the local attributes. Finally, the mapping view has *import* and *export* methods that access the appropriate local attributes. The SIT should be able to link the mapping and local classes and develop the *import* and *export* methods without needing information from the SIAT members. During the map step, the SIAT members provide the information the SIT needs to map the mapping view to the appropriate global classes. The example in Figure 10 will be used to describe the mapping process.

39

First, each mapping class is linked to every global class containing data needed by the mapping class. Each mapping class should be linked to at least one global class, but may be linked to more than one. In Figure 10 the *MapAircraft* needs to be linked to the *GblVehicle* class to obtain name information and to the *GblLocation* class to obtain location information. The SIT should provide a way for SIAT members to identify global and mapping classes to be linked.

Once all of the class linking is completed, all of the basic attributes of each mapping class should be linked to attributes of linked global classes. A global class must be linked to a mapping class if one of its attributes is linked to an attribute of the mapping class. Although the mapping class has no attributes other than the local and global classes to which it is linked, the SIT should make it appear as if the mapping class has the same attributes as the corresponding local class. For Figure 10 the *MapAircraft* should appear to have name and location attributes. Mapping these attributes to global attributes provides the information needed for the SIT to build the *get* and *set* methods of the mapping view. However, the mapping only represents a transformation of global data to the local format. There is no actual "link" to the global attributes. *MapAircraft's name* attribute maps to *GblAircraft's name* attribute. This is a straightforward mapping. *MapAircraft's location* attribute maps to an equation involving all three of *GblLocation's* attributes. For this case the equation $(xPos^2 + yPos^2 + zPos^2)^{1/2}$ translates a three coordinate position into a distance from origin. This represents a heterogeneity conflict called an attribute isomorphism conflict. Attribute isomorphism conflicts and their resolution are discussed in Section 3.3.

After the classes have been linked and the basic attributes have been mapped, the aggregation is mapped. Since the global and mapping schema are not required to have the same aggregate structure, the SIT may not always be able to correctly infer how to build a mapping schema over a global schema. Therefore, for each component attribute of each mapping class, an aggregation map must be provided by the SIAT. An aggregation map consists of multiple mapping sequences, one for each global class mapped to the component attribute class. Each mapping sequence shows

40

how to traverse the global hierarchy from a global class referenced by an aggregate mapping class to a global class referenced by one of the aggregate mapping class' components. A mapping sequence is made up of elements consisting of a global class name, one of its component attribute names, and the component attribute class name. The first element of each mapping sequence must contain the global class name of one of the global classes mapped to the aggregate mapping class. The next element must contain the same global class name contained by the previous element as the component attribute class name. The final element of the mapping sequence must contain an attribute class name that is one of the global classes mapped to the mapping component attribute class. This information is used by the SIT to generate the methods of each mapping class for generating local views of global data, importing local data into the global format, and exporting the global data into local formats.

For Figure 10 the component set of *MapWeapons* classes must be mapped. Figure 11 adds the component and mapping view attribute names to the example in Figure 10. It also simplifies the figure by showing only the mapping and global classes involved in the *hasWeapons* aggregation map. This component attribute would have one aggregation map. The aggregation map provides the information needed by the *MapAircraft* class to assign the proper value to the *viewOfGblResource* attribute of the *MapWeapons* class. The *MapAircraft* class is linked to the global *GblVehicle* and *GblLocation* classes. The *GblResource* class is a component of the *GblVehicle* class. The aggregation map must identify how to get from the *GblVehicle* class to the *GblResource* class needing to be linked to the *MapWeapons* class. The dotted circles and lines of Figure 11 identify the needed information. The value of the aggregation map would be: *GblVehicle:hasResources:GblResource*. There is only one mapping sequence in this map and it only has one element. The element begins with the global class linked to *MapAircraft* and ends with the global class linked to *MapWeapons*. This meets the requirements stated above for mapping sequences of aggregation maps. In this case the component attributes are collections. The

41
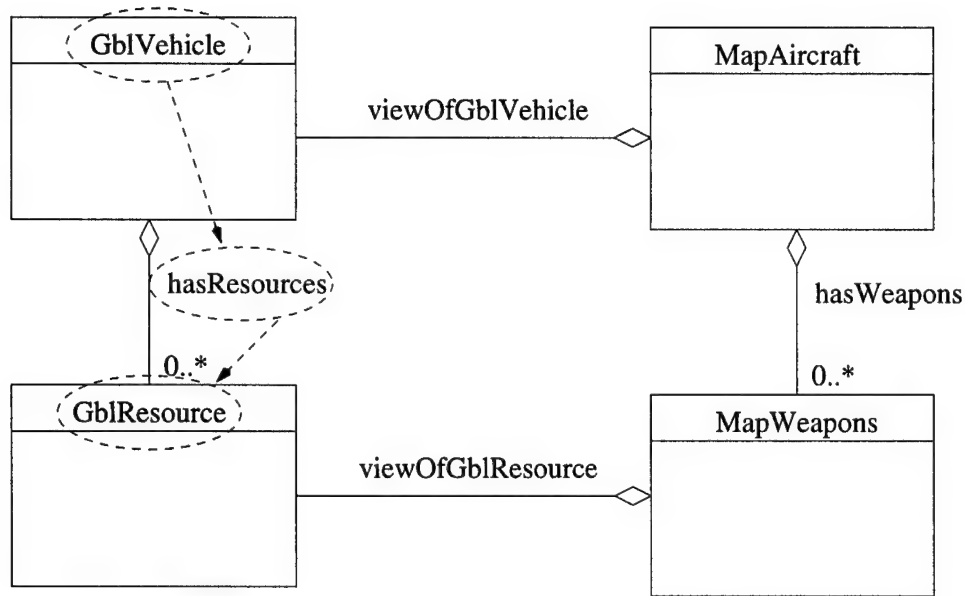
Figure 11.   Example of an Aggregation Mapping

SIT should be able to handle mapping for both collection and single component attributes.

Step 4: Finalize

This step is accomplished by the SIT. The SIT should provide an option that causes this step to occur. When this option is selected, the SIT finalizes all mapping information, generates the final attributes of the mapping schema, and generates the methods for the mapping and global schemas. Once a global domain is finalized, it can still be modified, integrated with additional local schemas, and finalized again.

Step 5: Generate Code

This step is also handled automatically by the SIT. The SIT will provide at least one code generation option, and may provide many for different languages. Multiple options may also exist for the same programming language, which generate the schema code in varying formats or with extra abilities for use with specific end-user tools.

1. Naming

    (a) Class

    (b) Attribute

2. Attribute Domain Definition

    (a) Data Type

    (b) Data Expression Representation

    (c) Data Units

    (d) Data Precision

    (e) Attribute Default Value

    (f) Attribute Integrity Constraints

3. Class Definition

    (a) Attribute Isomorphism

    (b) Missing Attribute

    (c) Attribute Composition

    (d) Class Invariant Constraints

4. Schema

    (a) Missing Class

    (b) Generalization & Inheritance Hierarchy

    (c) Aggregation & Aggregation Hierarchy

Figure 12.   An Enumerated Classification of Heterogeneity Conflicts

3.3 Heterogeneity Conflicts

When integrating two schemas, heterogeneity conflicts will almost always occur. This section describes the general types of conflicts and what the SIAT members' and SIT's roles are in resolving them. The heterogeneity conflict classifications, shown in Figure 12, are based on a modified combination of the classifications given in Misser, Rusinkiewicz and Jin's work [8] and Kim and Seo's work [19:521-550] [20]. Both classifications tend to focus on integrating relational database (RDB) schemas with RDB or object-oriented database (OODB) schemas. This work's classification and resolutions are based on using the traits of

mapping schemas and the abilities of the SIT to integrate object-oriented (not necessarily database) schemas. This results in some shifting of categories and changing the resolution techniques. Colonese's classification [6], based on a subset of Kim's work, was also used as a basis for this work's classification.

Each of the conflict classification areas, with their corresponding resolutions, are discussed in the following sections. Resolution of heterogeneity conflicts is the main driver of complexity in developing the SIT. Therefore, each section discusses the possibility of disallowing certain actions in the conflict resolution to allow for a simpler SIT. As mentioned in Section 3.2, the design of the SIT should be kept as simple as possible to minimize time, cost and effort. If more capabilities are found to be needed, they can be incrementally added or worked around for a one-time need.

3.3.1 Naming Conflicts. Naming conflicts occur when classes or attributes in each of the schemas represent the same thing but have different names, or represent different things but have the same name. Identifying naming conflicts is very difficult to automate, but not difficult for schema experts to identify.

The nature of the mapping schemas, the SIT, and the integration process make naming conflicts trivial to resolve. Naming conflicts caused by different classes or attributes having the same name may be ignored. The SIT only associates things through the mapping schemas. If there is no link in the mapping schema, it is assumed none exists. Since all classes and attributes must be linked through the mapping schema, naming conflicts where classes or attributes represent the same thing, but have different names, are handled the same as any other linking. The SIAT member identifies which classes and attributes should be mapped and the SIT handles the mapping schema generation.

Since naming conflicts are taken care of automatically through the regular integration process, the SIT should be fully capable of handling this conflict. This is not difficult, since it requires no additional capabilities.

3.3.2 Attribute Domain Definition Conflicts. Attribute domain definition conflicts occur when attributes represent the same thing, but the attributes' domains do not

44

match, or are not used the same. These conflicts are caused by data types, units, expression representations, or the precision of the attributes' domains not matching.

3.3.2.1 Data Types. These conflicts occur when the data types of attributes do not match. If the conversion between domains is provided automatically by the programming language(s) desired for final code generation, then it can be taken care of by the SIT generating *gets* and *sets* using type casting. An example would be a real global attribute mapped to an integer local attribute. Both Java and C++ provide automatic conversion from reals to integers and integers to reals through type casting. Note that real to integer transformations are normally done by truncating at the decimal point, and cannot be reversed. When using automatic conversion the global schema should always be adjusted so that the global attribute has the broadest attribute domain. This ensures the transition from global to local always has sufficient information. If no automatic transition is available, conversion methods must be provided to convert between domains. Conversion methods may also result in data conversions that lose information. Problems with lost or missing data should be handled by end-tools or end-users.

3.3.2.2 Data Expression Representation. Data expression representation conflicts occur when different scalar values are used to represent the same data. For example, consider a vehicle class with a transportation Mode attribute. In various domains the values may be air, land, water; a, l, w; 1, 2, 3 (representing air, land, and water); flight, drive, float. The difference may be from different codes or abbreviations representing the same thing, or different expressions with the same meaning.

These conflicts can be resolved by using conversion methods. The conversion methods operate as static look-up tables. Each method accepts a value from one domain and returns a semantically equivalent value from another domain.

3.3.2.3 Data Units. These conflicts are caused by numerical domains being used to represent the same thing but using different units. The different units result in the same data carrying a different meaning. For example, centimeters and inches can both be integers (or subtypes of integer), but 5 cm is very different from 5 in. This presents

an added danger of values actually being accepted from one schema by another, and the difference in unit meaning causing strange results to occur during data operations.

Data unit conflicts can be solved by using conversion methods. The conversion methods usually consist of a numerical equation that converts between units.

3.3.2.4   Data Precision.   This type of conflict results from domains with differing cardinalities. For example, consider two airplane classes that represent the same concept in two different schemas. Both airplanes have a speed attribute. One attributes domain consists of slow, regular, and fast; while the other consists of integers from 0 to 1000. Both domains represent the speed of the airplane but many of the values of one domain map to one value in the other domain.

The solution for these conflicts is to provide conversion methods that provide a one-to-many mapping between domains. Going from the larger to smaller is valid, but results in lost information. Going from the smaller to larger domain requires some arbitrary legal value from the range to be selected. In this case the global domain should use the larger domain to preserve as much information as possible.

3.3.2.5   Attribute Default Values and Integrity Constraints.   Default value and integrity constraint conflicts are special cases of the domain definition conflicts and are caused by corresponding attributes in the local and global schema having different default values or integrity constraints. These conflicts are resolved by not allowing default values or integrity constraints for global attributes. If a global attribute is unset, exporting the data to a local schema will result in the default value for the local schema being used. However, when data is imported from local schemas, the default value for that local schema will be imported. This value may be undesirable for exporting to other local schemas. This problem should be handled by the end-tools or end-users. Similarly, values allowed for one local schema may not be allowed by another local schema, but the unconstrained global schema will accept all of these values. Again preparing the data for export to a particular local schema should be handled by end-tools or end-users.

3.3.2.6   SIT Requirements.   Since the goal is for the SIT to be able to generate the final code in multiple languages and formats, it is necessary for the SIT to do most of the work correcting attribute domain definition conflicts. All of these conflicts can be resolved by automatic type conversion (casting) or through a conversion method. The SIT should provide a choice for the user to select automatic type casting or method conversion. Automatic type casting should be the default since it is the most likely to occur, and can be handled automatically by the SIT. However, SIAT members must use caution in using this choice since which types have type casting between them available is programming language dependent. The SIAT members should be sure all desired target languages for code generation support the needed type casting before using this option.

If method conversion is selected, preferably the SIT will query the user for the necessary data and automatically transform the data to create the conversion methods. In most cases this should work; however there can be so many different kinds of conversion methods required, it becomes more feasible to have the SIT provide a way for the SIAT members to enter the methods directly. When direct method entry is needed, the SIT should require a generic format to be used for the method, enabling the method to be converted to equivalent methods in the desired code generation languages and formats. It is important to keep in mind that data will be both imported to the global schema from local schemas, and exported from the global schema to local schemas. Therefore, a sufficient number of conversion methods must be provided to handle conversion in either direction.

Some resolutions require the ability to modify the types or domains of global attributes. The global schema must have the ability to store the information needed by all local schemas. Therefore, the global attributes must use the broadest types and domains needed. The SIT must be able to support this, and to do so while keeping existing mapping schemas up to date. This may require conversion methods to be added to support already integrated local schemas. The SIT should at least provide a way for SIAT members to return to existing mapping schemas and perform the method/data input. If it doesn't greatly complicate the SIT code, it might be desirable to have the SIT immediately prompt the SIAT members for the conversion methods without having to return to the existing

mapping schemas. However, such ability should be added in addition to, not in place of, doing it later. Since it involves one or more other local schemas, the needed schema experts may not be available. Doing it later allows for the current integration to continue without having to wait for other SIAT members to become available.

This is an area where limiting what conversions can be done or what domain definition conflicts can be handled can greatly simplify the design of the SIT. SIT abilities in this area should be minimized and increased only when regularly required.

3.3.3 Class Definition Conflicts. Class definition conflicts occur when classes represent the same entity, but their attributes or the constraints on their attributes do not match. These conflicts include attribute isomorphism, missing attribute, attribute composition, and class invariant constraint conflicts.

3.3.3.1 Attribute Isomorphism. This conflict occurs when the same concept is defined by two classes but with a different number of attributes. This can be both a one-to-many or a many-to-many conflict. These conflicts can be solved by using conversion methods that act as a look-up table. The conversion methods should take one or more attribute values from one schema and return a value for an attribute in the other schema. For many to many conflicts there would be one method for each participating attribute. The method would return a value for the respective method and receive values for each of the attributes from the other schema as parameters.

A special case of this type of conflict is string concatenation conflicts. This occurs when a value is one string attribute in one schema and multiple string attributes in the other schema. For example a *name* attribute in one schema could correspond to *firstName* and *lastName* attributes in another schema. This conflict can be resolved by providing one method that concatenates multiple strings and returns them as one string, and one method for each of the multiple strings that extract the appropriate strings from the combined string.

3.3.3.2 Missing Attribute. The missing attribute conflict occurs when the global class does not have an attribute or set of attributes corresponding to a given local

attribute. This is easily resolved by adding an attribute to the global class. This has no effect on previously integrated mapping schemas, since none of them map to the previously non-existent attribute.

Sometimes a missing attribute's value may be inferred from other aspects of the corresponding schema than attributes. For example, an attribute of engine type (propeller or jet) for an airplane class may be inferred from whether an object is an instance of a Jet Plane or Propelled Plane class. For these types of cases the missing attribute conflict should be handled through the conversion method resolution as used in the attribute domain definition conflict resolutions.

3.3.3.3 Attribute Composition. These conflicts occur when a basic attribute (its domain is a basic type) in one class and a component attribute (an attribute whose type is another class) in the corresponding class represent the same data. With these conflicts, the global class should have the component attribute to provide the ability to maintain the most information. Then the conflict may be resolved in one of two ways.

First, the global component attribute class may be linked to the mapping class and the basic mapping attribute linked to a basic global attribute that represents the component global class. This should be done with caution, since it can hide the fact that the true relationship is between the mapping class and the aggregate global class, not the mapping class and component global class.

The second method is recommended, but drives more complexity into the SIT. For this method, the mapping basic attribute and component global attribute are linked. Then conversion methods are used which return or set the value of a selected basic attribute of the component global class when the value of the basic mapping attribute is requested or set.

Both resolutions result in data loss when going from the global to mapping schema. This is not a problem since the data is still stored within the global data store and unneeded by the local data store. The resolutions also result in missing data when going from the mapping to global schema. This is a more significant problem and should be resolved by end-tools or end-users providing the missing information.

3.3.3.4 Class Invariant Constraints. Class invariant constraint conflicts are caused by classes that represent the same entities having different invariant constraints. As with attribute integrity constraints, the solution is to not allow invariant constraints in the global schema.

This may seem extreme, since it would be reasonable to expect the invariant constraints to be equivalent or compatible often, if not most of the time. By not allowing invariant constraints in the global classes, data may be entered that is not permitted by any local schema. This is true, but there will always be potential data problems for export to the local schemas. Rather than complicating both the integration and data export preparation tasks, all data incompatibility problems should be handled by the end-tools and end-users during data export preparation.

3.3.3.5 SIT Requirements. Class definition conflicts are solved through conversion methods or the addition of attributes. The SIT requirements for conversion methods were discussed in Section 3.3.2.6. The need to add attributes is likely to occur often and has no effect on previously integrated mapping schemas. Therefore, the SIT should have the ability to add attributes to classes in the global schema.

3.3.4 Schema Conflicts. Schema conflicts cause the most difficulty and complexity in conflict resolution, updating previously integrated mapping schemas during conflict resolution, and generating view initialization, data import, and data export method designs and code. Schema conflicts occur when when the global schema does not represent entities that are represented in a local schema, or when the aggregation or inheritance structures of the global and local schema do not match.

3.3.4.1 Missing Class. The missing class conflict occurs when the global schema does not have a class or set of classes that corresponds to a given local class. The data represented by the local class is not present in the global schema. If the data is present but distributed among many classes, the conflict is actually a generalization or aggregation conflict.

The missing class conflict is resolved by adding a class to the global schema. This has no effect on previously integrated mapping schemas, since none of them map to the previously non-existent class. This does require the SIT to have the ability to add classes to the global schema. The necessity to add classes is likely to occur a lot and is not very difficult or complicated to add. Therefore, the SIT should have the ability to do this.

A new class should not have the same name as existing classes or types. Doing so will cause unpredictable responses from the SIT (or require a lot of effort and complexity to deal with), and will result in errors in the generated code.

When a class is added to the global schema, it is added as a leaf node of, or separate from any existing inheritance and aggregation trees of the global schema. Needing to move the new class to different positions in the aggregation or inheritance trees corresponds to aggregation hierarchy or inheritance hierarchy conflicts.

3.3.4.2 Generalization and Inheritance Hierarchy.   Inheritance hierarchy conflicts occur when the inheritance hierarchy of the global and local schemas do not match. A specific case is the generalization conflict, which occurs when one schema's class is a subclass of the other schema's class. For example, one schema may have an airplane class while the other schema has an F-16 class. This is referred to as an inclusion conflict in Kim's work [19], since one class includes all the instances of the other class. Other inheritance hierarchy constraints can usually be broken down into multiple instances of the generalization or other conflicts.

The SIT shows all attributes, inherited and otherwise, for each class, and all attributes are mapped. The goal is to transfer the data in the classes, not the inheritance hierarchy. Therefore, as long as the global schema has all of the attributes needed by the local schemas, it does not matter if they are in one superclass or a superclass and many subclasses. However, attributes may need to be added to insure the global schema has all of them needed by each class.

It might be beneficial to reorganize the global schema's inheritance hierarchy to make it easier to enter data directly into it, or to better facilitate the processing of data for exporting to local schemas. In this case the inheritance hierarchy conflict may be solved by

solving the instances of the other conflicts that make it up. Generalization conflicts may be solved by adding the missing subclass or superclass and redistributing the attributes between them. As long as these changes are occurring to leaf nodes of the inheritance hierarchy, it is not very difficult to have the SIT track the changes and update the existing mapping schemas. However, if the changes occur higher in the inheritance hierarchy, it can cause the loss and addition of many attributes to many classes. It could also cause changes in the aggregation hierarchy. These potentially extensive effects can make it very difficult for the SIT to automatically update the existing mapping schemas. Therefore, changes to the inheritance hierarchy, other than leaf nodes, should be avoided unless needed to meet the goals of the integration. Further, giving the SIT the ability to handle these inheritance hierarchy changes should only be done if they are going to occur often.

3.3.4.3 Aggregation and Aggregation Hierarchy. Aggregation hierarchy conflicts are caused by the aggregation hierarchies of the local and global schemas not matching. The attribute composition conflicts described in Section 3.3.3.3 are actually the simplest form of aggregation hierarchy conflicts. Aggregation conflicts are really just a combination of one or more attribute composition conflicts. They occur when a concept is modeled as a class in one schema and an aggregation of classes in the other. An example would be a class consisting of a collection of weaponry with attributes that describe the traits of the collection versus a class consisting of the same collection of weaponry with an attribute that is a set of weapon classes.

The more complicated aggregation hierarchies are conflicts involving different numbers of classes representing the same concept in each schema. These conflicts are made of multiple attribute composition, aggregation, missing attribute, and attribute isomorphism conflicts.

As with inheritance hierarchy conflicts, there are two ways to deal with aggregation hierarchy conflicts. First, the use of aggregation maps allows the aggregation structures to remain different. Aggregation maps identify to the SIT how the mapping schemas aggregation hierarchy can be derived from the global schema hierarchy. Second, aggregation hierarchy conflicts can be broken down into the lower level conflicts, which can

52

each be resolved. Interpreting aggregation maps can be very complicated, especially when the map passes through multiple collection component attributes. However, the combination of all the conflict resolutions needed to solve an aggregation conflict can also become very complicated for the SIT to track and use to update previous mapping schemas. The best approach is to use a combination of aggregation mapping and conflict resolution to maximize achievement of integration goals and minimize complexity of the SIT. It is not recommended to just disallow one of the two methods completely. The conflict resolutions are needed to handle the lower level conflicts regardless of their use with aggregation hierarchy conflicts. Aggregation maps can facilitate local schemas that only map to a branch of the global schemas aggregation hierarchy, and not the top.

This two-method approach is different from existing integration methodologies which focus on solving the lower level conflicts. The use of the two method approach gives more flexibility and capability to both the SIAT members and the SIT during integration and code generation.

3.3.4.4   SIT Requirements.   Schema conflicts are the most complicated conflicts to solve and are often made of several other conflicts. Therefore, the SIT requirements were generally discussed with each of the schema conflict types. However, there are a couple of additional notes. Inheritance and aggregation hierarchy conflicts both have dual methods for solving. It is important that the SIT's ability for handling conflicts and mapping inherited attribute and aggregation be considered together during implementation and not on a one-by-one basis. The SIT needs to be able to handle combinations of resolutions, not just independent solutions.

Resolving inheritance and aggregation conflicts through resolving the sub-conflicts may require deleting classes or changing their positions in the aggregation or inheritance hierarchies. The SIT should handle these changes in one of four ways:

1. Allow the changes, but query SIAT members for additional information to provide re-mapping of affected attributes and aggregations of previously integrated mapping schemas.

2. Allow the changes, but issue a warning identifying which attributes and aggregations of previously integrated mapping schemas need to be re-mapped.

3. Allow the changes, but issue a warning identifying which changes were not checked for effects on previously integrated mapping schemas.

4. Disallow the changes.

The four items are listed from the most desirable option to the least desirable option. However, the more desirable the option is, the more complexity it adds to the SIT. A mix of the solutions should be selected that balance maximizing integration goals and minimizing SIT complexity. If lower options are selected, the SIT can always be modified later to use the higher options.

3.4 Summary

This chapter described the data format translation problem. It also provided a methodology for solving the problem through integrating data store schemas into a global schema. The methodology is based on object-oriented and transformation system technologies. Integration is performed by the combined effort of an integration tool and a team of schema experts.

The methodology emphasizes flexibility and automation. The flexibility and automation are provided by varying the capabilities of the integration tool. This allows human involvement in the methodology to remain basically unchanging. The capabilities of the integration tool are increased as necessary to meet the goals of the integration, while minimizing the complexity of the tool as much as possible. This minimizes development and maintenance time, effort, and cost of the integration tool.

The methodology provides dual approaches to solving inheritance and aggregation hierarchy conflicts. This allows for greater flexibility in solving the conflicts. This in turn allows the resolution of more complicated conflicts while:

- Not requiring the aid of a database.

- Minimizing the complexity of the integration tool.

- Allowing integration of local schemas that only map to sub-branches of the global schema's aggregation hierarchy.

The next chapter presents an implementation of the SIT and illustrates the steps of the methodology through the integration of battle simulation systems.

IV. Application of Methodology to Battle Simulation Systems

There is a great deal of flexibility to the methodology described in Chapter 3. Most of this flexibility is based on the development, capabilities, and limitations of the SIT. This chapter presents one implementation of the methodology. The implementation is intended to demonstrate the feasibility of the methodology, not to further define or restrict it. Design decisions made in the implementation are based on the criteria defined in the methodology, the time considerations of this effort, and occasionally to facilitate the demonstration of particular traits and flexibility of the methodology. The chapter is organized into eight sections. The first seven sections describe the execution or implementation of the two preparation and five regular steps of the methodology. The chapter ends by describing how to use the code generated by the SIT to develop end-tools.

4.1 Preparation Step 1: Goal Determination

To implement the methodology, there must be something to integrate. As discussed in Chapter 1, the Sensors Division of the Air Force Research Laboratory is greatly interested in the integration of sensor-based engagement-level simulation systems. Therefore, such simulation systems are used in this implementation. More specifically, Suppressor Composite Mission Simulation System [6, 33, 35, 38] (Suppressor), Simulated Warfare Environment Generator [22, 38] (SWEG), and Extended Air Defense Simulation [6, 36, 37] (EADSIM) are considered.

Suppressor is a mission-level simulation that models multi-sided conflicts comprised of air, ground, naval, and space forces. Suppressor uses seven different input files, of which four are required, one is recommended, and two are optional.

SWEG was originally derived from Suppressor and is also a mission-level simulation. SWEG uses nine input files, eight containing information similar to the seven Suppressor input files, and an additional file used to set up symbols and colors for graphical output.

EADSIM is a simulation of air and missile warfare. Each platform is individually simulated along with the Command and Control decision processes, the communications among platforms, and intelligence gathering. EADSIM provides the Scenario Generator

application, which provides tools for scenario definition, modification, and analysis. The Scenario Generator saves the scenario data into 11 input files, used to initialize the scenario in the run-time modules, which perform the simulation.

The input file of interest from both the Suppressor and SWEG simulations is the Type Database (TDB). The major component of the TDB is the *Player*. *Players* represent the entities used in the simulation. It is the *Players* and the interactions between the *Players* that is being simulated. Examples of *Players* could be a bridge, a jet fighter, a person, or even a swarm of bees. EADSIM has similar data stored in its laydown and element files. These files define *Platforms*, which are EADSIM's equivalent of the *Players*.

Both Weber and Colonese focused on integrating sections of the *Player* (*Platform*) structure. Weber integrated high level models of the *Player* structures in Suppressor and SWEG. Colonese integrated the *Airframe* aggregate EADSIM *Element* with the Suppressor *Mover* (helicopter and airplane) *Resource* structure. Colonese's integration was a much smaller one done at a much higher level. However, Colonese integrated the aircraft of EADSIM with the aircraft of Suppressor. This brings up an interesting issue. With EADSIM's Scenario Generator, the type of *Platform* and *Element* being modeled is selected before data is entered. For example, an airplane is selected, and then the proper data fields are displayed to the user for input. Colonese's integration demonstration was performed at this level of detail. She knew that she was integrating aircraft with aircraft. Weber's model of the *Player* structures of Suppressor and SWEG follow the TBD file structure. To determine if a *Player* is an aircraft requires examination of the capabilities of the *Player*. Analysts who wish to use Suppressor *Player* data in EADSIM simulations will probably have certain aircraft or missiles in mind and will be much happier if they are displayed by the model as aircraft and missiles as Colonese's models and integration do. However, models like Weber uses, which follow the non-object-oriented Suppressor and SWEG TDB file structures, are much less complicated to develop. This is a trade-off that should be discussed during this step and the following one. Whether models are more representative of the way data is stored, or the way end-users would use the data should be based on the goals for integration, which are developed during this step.

57

The EADSIM model used by Colonese is the only object-oriented model available for EADSIM. However, the level of detail currently used would not provide an interesting test case or fully test the methodology. The level of detail available in Suppressor and SWEG's object-oriented models is much greater and provides a more complete test of the methodology. For this demonstration it is desired to fully integrate two local formats, finalize the integration, and generate code. Then, after integrating a third local format, the integration finalizing and code generation are repeated. This demonstrates the ability of the methodology to handle the integration of additional local formats after the initial integration has been completed and finalized. Suppressor and SWEG provide examples of "real world" formats against which the methodology may be demonstrated. Therefore, rather than trying to develop more detail into the EADSIM model, it was decided to develop an example of a possible simulation system with an appropriately detailed model. The model was developed in such a way as to ensure that all elements of the methodology were fully tested. This allowed for both "real world" and more complete developmental testing to be accomplished. The "made-up" simulation's name is ASHSIM. ASHSIM simulates battles between battle groups. Examples of battle groups are squadrons of aircraft, fleets of ships, and battalions of soldiers.

There are two main goals of this integration. The first is to demonstrate all aspects of the methodology. The second is to integrate ASHSIM's *BattleGroup* structure with portions of Suppressor and SWEG's *Player* structures. This results in an integrated *Player* structure (containing battle groups also) that can provide *Player* data for all three simulation systems, and transfer *Player* data between systems.

4.2 Preparation Step 2: Develop Models

Next, object-oriented models are developed for each of the data stores to be integrated. The Type Databases for Suppressor and SWEG are flat files. Therefore, they not only need object-oriented models, but parsers for transferring data between the object-oriented data models and the flat files. Since ASHSIM is being "made-up" for this validation, it is developed at the object-oriented model stage.

58

The goals to develop an integrated *Player* structure that can provide *Players* for all three integration systems, and to demonstrate the methodology should be kept in mind during these developments. It may be desirable to process the Suppressor and SWEG data in order to identify the types of *Players* being stored. For example, airplanes, helicopters, and missiles would be valuable players for EADSIM to use. This is why it is so important to have data store experts available from all simulation systems. Having the object-oriented model based on the structure of the flat files does not necessarily provide the best-suited model for integration and reuse. In fact, it may not even provide objects comparable to the modeled real world objects. Since the global model is translated to other data stores before use, it is not required for the objects to make sense compared to the real world objects for the process to work. However, such correspondence would make it much easier for analysts to enter in data or identify desired *Players* for their simulation systems.

Although the above cautions are provided, the actual development of object-oriented models and the parsers between them and their non-object-oriented formats are out of scope for this effort. Therefore, ASHSIM is minimally developed, and modified versions of existing object-oriented models for SWEG and Suppressor are used. The SWEG model developed and used by Weber for his thesis [38], and the Suppressor Model developed and used by McDonald in his thesis [23] are used. All three models are displayed in Appendix B.

4.3   Step 1: Prepare the Schema Integration Tool (SIT)

4.3.1   SIT Developed Using AWSOME and Java Swing.   For development of the SIT, three products were considered: Rational Rose, AFITtool (REFINE based), and AWSOME (Java Based AFITtool). Both AFITtool and AWSOME are transformation systems with the ability to store, manipulate and generate code for software designs, including data store schemas. However, both are still being developed and have only achieved partial functionality. Rational Rose is not a transformation tool, but a modeling tool. It provides a nice GUI interface for entering and modifying schemas and some code generation. Through its extensibility features, it can be extended to allow mapping between classes and attributes, and provide more extensive code generation. It is also widely used by industry. Since the AFITtool and AWSOME systems are still under development and

Rational Rose is not a transformation tool, any of the three choices would have required development work to be completed. For this effort, much more access and control of the code was available for the AFITtool and AWSOME systems. Also, other efforts were ongoing to increase the capabilities of AFITtool and AWSOME to make them more complete. AFITtool and AWSOME also required less additional training to use since training had already been provided in their use and development. Therefore, it was decided that either AFITtool or AWSOME would be used for this effort.

One of the main goals of the methodology is to allow for use of commercial software, while providing the flexibility of switching which commercial products are used, or not using any at all. This better supports the needs of the user and better handles the commercial market, both of which are constantly changing. This is another reason AFITtool and AWSOME were chosen over Rational Rose. The REFINE-based AFITtool relies on the commercial Software Refinery tools, which are expensive and not common in industry. AWSOME only relies on the Java programming language, which is freely available. Also, AWSOME development had just been recently initiated, which allowed input to be provided in its development to insure the required capabilities were available for this effort. Therefore, AWSOME was chosen as the foundation of the integration tool used in this effort.

AWSOME had no available user interfaces. Therefore, one had to be implemented for this effort. One option was to use Rational Rose as an interface, as AFITtool is able to do. Rational Rose (as extended by Noe) [24] generates model information in Z LaTeX form, which can then be parsed into AFITtool. No parser is currently available for AWSOME, and there was insufficient time available in this effort to accomplish one. This is not a big concern, since this demonstration is intended to be a non-commercial product dependent implementation. A minimal Java Swing [25] application was used as the user interface for this effort. The main window of this tool is shown in Figure 13.

4.3.2 SIT Capabilities.   Following the caution of the methodology, the SIT was developed with minimal capabilities. The SIT has the ability to load a local schema from a Java serialized file or from a server. Once loaded, a local schema may not be changed in any
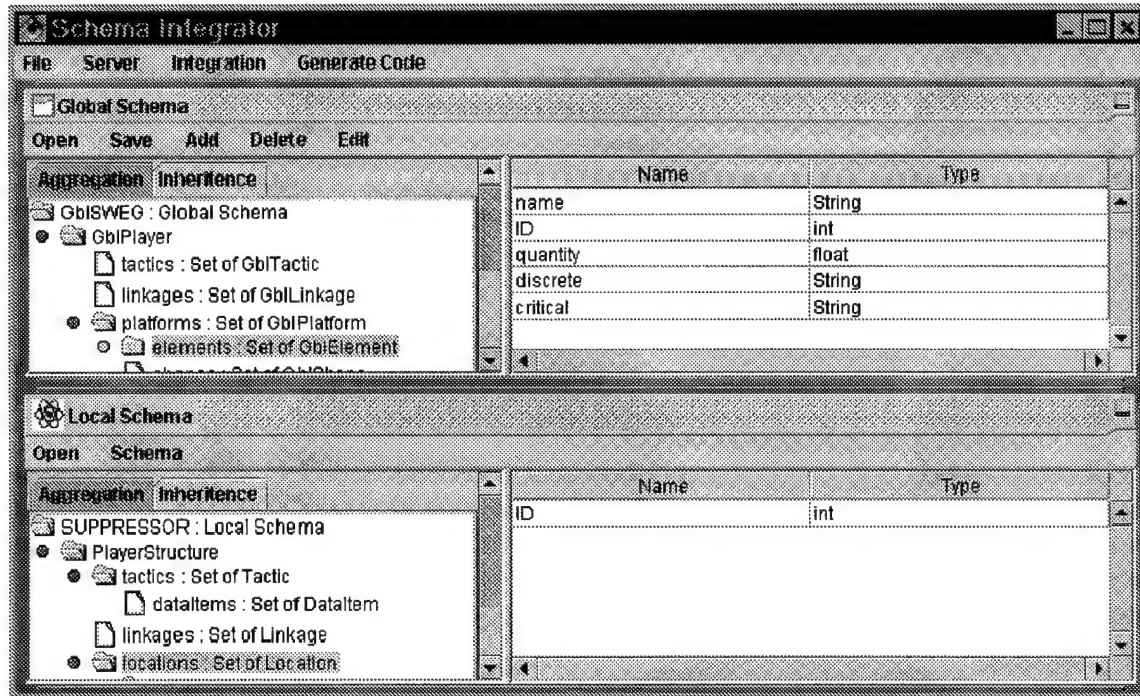
Figure 13.   The Schema Integration Tool (SIT) Main Window

way. This protects any tools that may have already been developed for the local schema. For example, it protects parsers that load data from the non-object-oriented versions of the local data stores from becoming incompatible with the local schema. A local schema may be viewed, converted to a global schema, or converted to a mapping schema. As called for in the methodology, the mapping schema appears to be an exact duplicate of the local schema, and references the corresponding local classes' attributes through *import* and *export* methods. As seen in Figure 13, the current global schema is displayed in the upper window, while the current local or mapping schema is displayed in the lower window.

The SIT provides the ability to link global and mapping classes, map attributes, and map aggregation. It also provides the ability to add, edit, or delete classes, attributes, and types in the global schema. The SIT will also update attribute maps in existing mapping schemas to reflect adding or deleting classes or attributes, and editing attributes. The SIT is unable to update aggregation maps of existing schemas, or to compensate for changes to a class' superclass. Further SIT capabilities, and lack of capabilities, along with their implications, are discussed in the next two sections on heterogeneity conflicts and mapping.

61

The final step in preparing the SIT is to initialize a global domain. This is done by selecting the broadest local schema (the one that contains the most common and extra classes and attributes) and converting it to the global schema. The conversion is done automatically by the SIT when the local schema is loaded and the "convert to global" option is selected. A mapping schema from the local schema to the new global schema is also developed automatically.

All three schemas contain classes not contained by the other two. However, SWEG has more classes in the aggregation hierarchy than either of the other two. Therefore, SWEG was chosen as the initial global domain. Suppressor was selected to be integrated to the global domain next, because it more closely resembles SWEG than ASHSIM does, resulting in fewer heterogeneity conflicts. ASHSIM was integrated last. It was specifically designed to cause some more interesting heterogeneity conflicts.

4.4 Step 2: Compare, Prepare, and Resolve

This step was performed separately for integrating Suppressor and integrating ASHSIM. For both integrations, many naming and attribute domain definition conflicts were found. There were also several class definition conflicts and a few schema conflicts. As the methodology calls for, the schema conflicts were examined first, followed by the class definition conflicts, and finally, the attribute domain definition conflicts. The resolutions used for these conflicts follow the methods presented in Section 3.3. Specific details about the conflicts and their resolutions are found in Appendix C.

There are a couple of schema conflicts that should be presented here. The first is an aggregation conflict between Suppressor's schema and the global schema. Suppressor has a collection of one or more instances of the *DataItem* class under the *Tactic*, *Susceptibility*, and *Capability* classes. The global *Tactic*, *Susceptibility*, and *Capability* classes do not have this; instead they have several attributes that attempt to store the same data. This is a fairly drastic difference in modeling the same type of information. A closer look at the non-object-oriented models on which the object-oriented models are based shows that the data is basically the same data and could be represented in both models the same way using either of the approaches used by the two models. The two models used by this effort

were developed independently with the same general goals, but differing specific goals. Had schema experts with the proper coordination actually performed the preparations steps, this conflict may not have occurred.

The varying number of allowed *DataItem* instances and the possibility of the instances which correspond to the attributes in global model classes not even occurring, or occurring in any order, made this a very complicated and difficult conflict to resolve. During the first non-trivial integration, it may be best to determine if the object-oriented version of either local schema may be changed without impacting existing local schema tools. During later integrations, the current local schema could still be considered. It was assumed this option was not available for this effort.

If the local schemas are unable to change, then the methodology must be applied to the conflict. The methodology calls for issues based on instance data to be handled by the end-tools and end-users. Therefore, the *DataItem* class was added to the global schema without affecting or removing any of the corresponding basic attributes. Since the *DataItem* collections were added as leaf nodes of the global aggregation hierarchy, attribute and aggregation maps of existing mapping schemas were unaffected. SWEG instance data will map to the basic attributes and Suppressor data will map to the component collection. This requires data to be processed by end-tools before data can be exported to SWEG or Suppressor format, which is not a very elegant solution.

The integration of ASHSIM resulted in a conflict with similar implications. In the global schema the *Element* class has a set of *System* class instances, each with a set of *Capability* class instances and a set of *Resource* class instances as components. In the ASHSIM schema there is a *SubGroup* class that corresponds to the global *Element* class. The *SubGroup* class has a set of *Vehicle* class instances and a set of *Weapon* class instances as components. These classes both correspond to the global *System* class and both sets should map to the single set of *System* class instances in the global schema. This conflict is further complicated by *Vehicle* having two subclasses. A closer look at the classes shows that both the *Vehicle* and *Weapon* possible class instances are a subset of the possible *System* instances. The basis for the subsets is the value of the *category* attribute of the global System class. The *Vehicle* and *Vehicle* subclass instances correspond to *System*

instances with *category* values of vehicle, transport, or combat vehicle. The *Weapon* class instances correspond to *System* instances with a *category* value of weapon.

Several ways exist to resolve this conflict. Which method is chosen should be based on the goals of the integration, determined in preparation step 1, and the capabilities of the SIT. However, no matter which method is chosen, data will still be coming from the local formats with and without *category* attributes. This data must be processed at some point between parsing in from one local format and parsing out to another local format to insure that the proper data is exported with the proper value to the target local format. This type of processing is dependent on instance data and should be handled by the end-tools or end-users.

This conflict is similar to the Suppressor aggregation hierarchy conflict involving the *DataItem* class, which also has the issue of requiring data to be processed by end-tools or end-users. As with the Suppressor conflict, this conflict may be due to a lack of coordination in designing the object-oriented models. However, this model has two sets of different, but similar, data classes whereas the global schema has one. Therefore, this may represent a very real need for the local schema format. The reason the ASHSIM schema was designed with this conflict in it is to demonstrate two points:

1. The importance of the two preparation steps, especially the importance of the object-oriented local schemas being developed in coordination with each other and the goals of the integration.

2. Even with the proper coordination between the development of the object-oriented local schemas, complicated schema conflicts may occur that require a combination of schema integration and data integration/processing to resolve.

The second point implies that there may be merit in developing a methodology that is able to combine the integration of data schema formats and data instances. This is discussed more in Chapter 5 as recommended follow-on research.

The SIT seemed to perform a minor role in this step. The schemas were compared, heterogeneity conflicts were identified, and resolutions were selected by the integrator using visual inspection. All the SIT provided was a graphical display of the two object models

64

being integrated. However, during this step, the global schema was modified by adding classes and attributes, and changing the domain of attributes. The SIT provided the ability to make these changes. Figure 14 shows the *window* used by the integrator to enter in the data needed by the SIT to create a new attribute. All changing, deleting or adding of data is done by providing data in similar *windows*, with the SIT handling the actual adding, deleting, or changing of model elements. Whenever the SIT was instructed to delete or modify existing elements, it checked the existing mapping schemas and identified affected mappings. The SIT then requested information from the integrator to update the affected mappings and updated them. This prevented a lot of re-integration work being forced on the integrator.
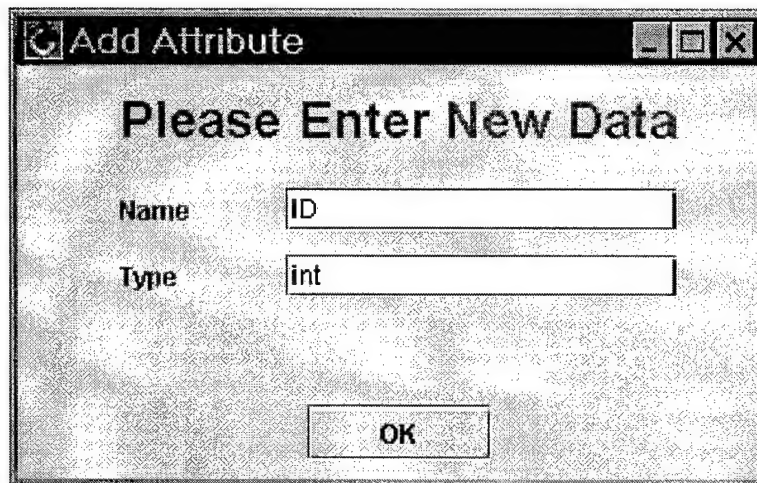
Figure 14.   The SIT Add Attribute Window

4.5   Step 3: Map

As with the compare, prepare, and resolve step, the map step was performed for both the Suppressor and ASHSIM integrations. The compare, prepare, and resolve step identified what classes and attributes should be mapped to each other and what heterogeneity conflicts prevent the mapping. Further, resolutions for the conflicts were also identified. Once identified, all resolutions were accomplished, except for naming conflicts and those

65

involving data type conversion and aggregation mapping. Naming conflict resolutions are handled through the regular mapping process and need no discussion.

The first task of this step is to link mapping classes to the corresponding global classes. This is done by selecting both a global and mapping class, and then selecting the "link classes" option from the integration menu. Once all the classes are linked, the basic attributes are mapped. To do this, a mapping class is selected and then the "link attributes" option is selected from the integration menu. The SIT prompts for a mapping attribute to be selected from a list, then for a linked global class to be selected from a list, and finally for a global attribute to be selected from a list. All lists are provided by the SIT and are determined based on previous selections. The *window* presented by the SIT to obtain this information is shown in Figure 15. The *window* is shown requesting the global class and will request the global attribute after the *ok* button is pressed. When the global and local attributes have the same data type, the SIT builds the aggregation map and the integration continues. When the data types do not match, the conversion method options selected is followed. The conversion method option is selected by *mouse clicking* on the desired option shown in Figure 15.

Data type conversions are needed for this integration. As recommended by the methodology, when attributes with different data types were mapped, the SIT prompts for automatic type casting or conversion methods. Automatic type casting is the default method and is handled by the SIT without further input. However, this is done through type casting regardless of whether a given language supports that type casting or not. Therefore, the SIAT member should be confident that any target languages for code generation support the casting. For this integration automatic type casting is selected for float to integer and integer to float. The Java and C++ languages support both of these, and are currently the only languages for which the SIT has code generators.

When conversion methods are selected, the SIT has several options for defining them.

- A text file can be provided, which the SIT reads in and parses into the proper AWSOME AST format for a subprogram.

66
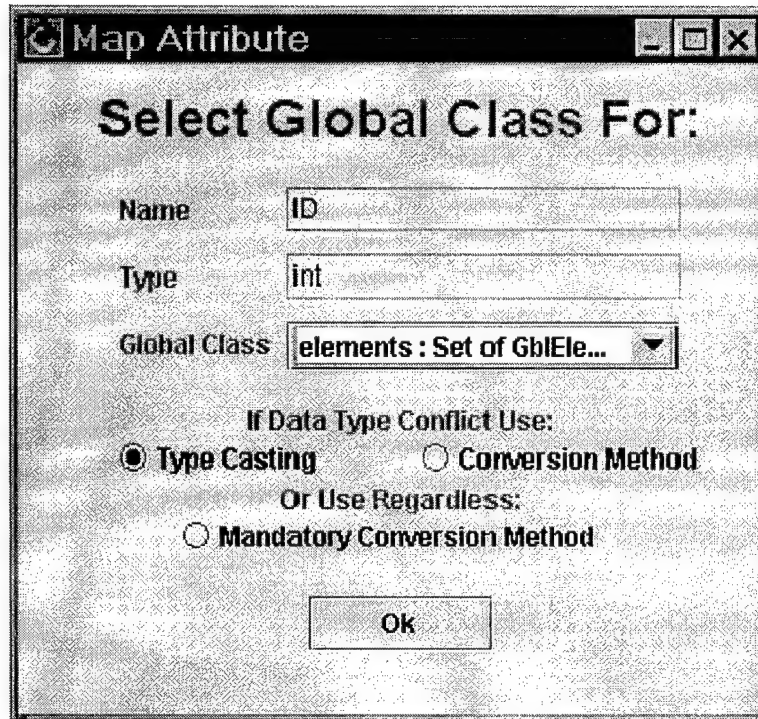
Figure 15.   The SIT Link Attribute Window

- Several common conversion methods are provided in AWSOME AST format and may be selected.

- An expression can be provided, which the SIT parses and uses to build a method.

The integrator selects the desired option from a *combo box* in the *window* shown in Figure 16. Once the method is selected, the SIT displays other data fields or windows to query the user for a text file name, provided conversion method name, or mathematical equation. The particular cases requiring conversion methods and how they were accomplished are discussed in Appendix C.

Since the SIT is unable to update aggregation maps, it is recommended to stop at this point and perform the above steps on all local schemas before proceeding. This reduces having to redo or adjust existing aggregation maps after each new integration. However, this demonstration is designed to show how local schemas identified for integration later can be integrated with minimal changes needed to existing mapping schemas. Therefore, the aggregation mapping was completed. The SIT does provide the capability to modify
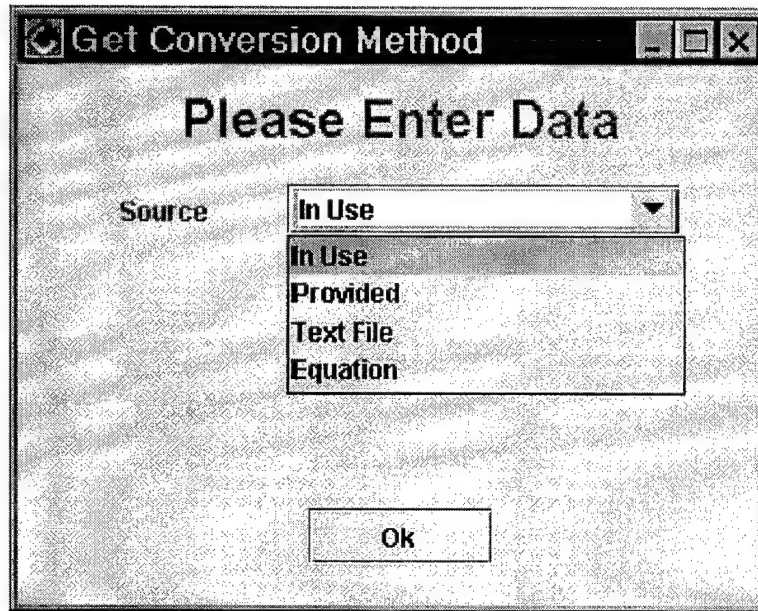
Figure 16.   The SIT Conversion Method Window

aggregation maps in existing mapping schemas, so any adjustments can be made without having to re-integrate. If more time had been available, the SIT would have been enhanced to perform some of the aggregation map updates automatically.

When the "map aggregation" option is selected, the SIT presents the *window* shown in Figure 17 to the integrator. The integrator selects a global class linked to the aggregate mapping class from the first *combo box*, and a global class linked to the component mapping class from the second *combo box*. Then the integrator enters the aggregation map as described in Section 3.2, Step 3.

4.6   Step 4: Finalize

The mapping information provided during Step 3 is needed to build the methods used by the mapping schema for accessing the global data (*get* and *set* methods), accessing the local data (*import* and *export* methods), parsing data from the local formats into the global format (*importView* methods), building mapping schema instances over global schema instances (*initView* methods), and outputting the global data in local formats (*exportView* methods). However, these methods are not created while the mapping is

68
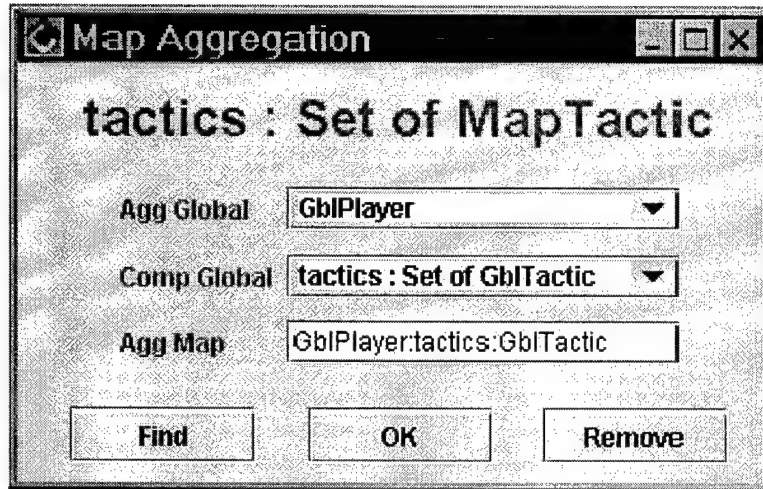
Figure 17.   The SIT Map Aggregation Window

occurring. During integration, the mapping information is stored in *IntegrationMap* and *AggregationMap* classes associated with the mapping and global schema classes. This format makes updating existing mapping schemas much easier to accomplish and minimizes the complexity of the SIT. The finalize step is accomplished by selecting the "finalize integration" menu option. Once "finalize integration" is selected, the SIT generates (or regenerates if "finalize integration" had been selected at least once already) all the above mentioned methods, along with attributes for the mapping schema classes that reference linked global classes. This is done for all integrated mapping schemas, and cannot be accomplished if any of the mapping schemas are only partially integrated.

4.7   Step 5: Generate Code

For the Suppressor integration, both C++ and Java code was generated for the global domain. The *importView*, *initView*, and *exportView* methods, discussed in Section 4.6, were not generated for this initial integration. The C++ code generator was only partially completed and was not able to handle the complexity of these methods. This code generation was only meant to demonstrate the ability to generate code in multiple languages. The code generation step for the Suppressor integration was accomplished by disabling the generation of the *importView*, *initView*, and *exportView* methods during the

69

finalize step, and then selecting the "C++" and "Java" options from the code generation menu.

For the ASHSIM, or final, integration, the generation of the *import View*, *init View*, and *export View* methods were not disabled. Selecting the Java code generation option completed the code generation step. C++ code generation was not selected since the code generator was incomplete. A sampling of the generated Java code is provided in Appendix D.

All code generated is fully compilable and ready for use by end-tools. Some end-tool is required, since none of the generated code does anything without being called by another program.

4.8   How to Use the Code Generated by the SIT

Originally, it was intended to develop a sample end-user tool to show how the code generated by the SIT may be used. This tool would allow data to be interactively entered into the global format, or parsed from files with data in local formats and then edited. The data in the global store could be viewed directly or through the mapping schemas for each local format. While viewing the global data through a mapping schema, the global data could be exported into the local format. The tool would have required the code to be generated in Java, but inheriting from some graphical classes and implementing some additional graphical methods, which are required to view the data graphically. The special inheritance and method generation can be accomplished through adding another code generator that not only uses the proper programming language but uses the proper format as well. This was done for this tool as the graphical Java option of the SIT's code generation menu. However, the time constraints of this effort did not allow for this tool or the graphical Java code generator to be completed.

The most important things the tool would have demonstrated would be how to use the generated mapping schema methods to import local data into the global store, generate local views of the global data, and export global data into local formats. These items are discussed in the next three subsections.

4.8.1 Parsing Data from the Local Formats into the Global Store. The *import-View* methods of the mapping schemas provide the local to global parsing capabilities. These methods will only take data from the local object-oriented format to the global format. However, when the local object-oriented schemas were developed, they should have been developed with the ability to parse data in from non-object-oriented local formats. For this discussion, it is assumed that each local aggregation root class contains a method named *load* that handles the parsing of local data into the local object-oriented format. With this assumption, the end-tool would accomplish importing local data into global format by performing the following steps:

1. Create an instance of the local aggregation root class (*LocalTop*).

2. Call the *load* method of *LocalTop*.

3. Create an instance of the mapping aggregation root class (*MappingTop*).

4. Call the *importView* method of *MappingTop*, passing *LocalTop* as a parameter. This *importView* method handles calling all of the other needed *importView* methods.

5. The *importView* method returns an instance of the aggregate global class (*GlobalTop*).

LocalTop now holds an aggregation hierarchy containing the local data in object-oriented format. *GlobalTop* contains an aggregation hierarchy containing the local data in global format. *MappingTop* is linked to both the local and global aggregation instances and can be used as a local view of the global data.

4.8.2 Generating Local Views of the Global Data. The *initView* methods of the mapping schemas provide the ability to generate local views of the global data. The end-tool would accomplish creating a local view of the global data by performing the following steps:

1. Create an instance of the mapping aggregation root class (called *mappingTop* for this discussion).

2. Call the *initView* method of the *mappingTop*, passing the global aggregation root of the desired global data (*GlobalTop*) as a parameter. This *initView* method handles calling all of the other needed *initView* methods.

MappingTop is now linked to the global aggregation instance and can be used as a local view of the global data. This is accomplished by all of the mapping schemas' *get* and *set* methods getting and setting data from the appropriate global classes and attributes.

4.8.3   Exporting global Data into the Local Formats.   The *exportView* methods of the mapping schemas provide the ability to export global data into local formats. These methods will only export data from the global format to the local object-oriented format. However, when the local object-oriented schemas were developed, they should have been developed with the ability to export data from the object-oriented local formats to the non-object-oriented local formats. For this discussion, it is assumed each local aggregation root class contains a method named *export* that handles the exporting of object-oriented local data into the local non-object-oriented format. With this assumption, the end-tool would accomplish exporting global data into a local format by performing the following steps:

1. A corresponding mapping schema instance for the local format must be linked to the global aggregation instance before the global data may be exported. If this has not been done, the step in Section 4.8.2 should be followed to accomplish it.

2. Create an instance of the local aggregation root class (*LocalTop*).

3. Call the *exportView* method of mapping aggregation root (MappingTop), passing *LocalTop* as a parameter. This exportView method handles calling all of the other needed *exportView* methods.

4. Call the *export* method of *LocalTop*.

LocalTop now holds an aggregation hierarchy containing the global data in local object-oriented format. The export method of *LocalTop* has created an instance of the global data in the non-object-oriented format.

4.9   Summary

The methodology presented in this effort is based on the use of a SIT. A sample development and implementation of a SIT was discussed in this chapter. A more de-

tailed description of the SIT implementation used for this demonstration is provided in Appendix A. The methodology was demonstrated using the SIT implementation and integrating three battle simulation schemas. Initially, two schemas were integrated with the resulting global, mapping and local schemas generated in C++ and Java. This demonstrated the capability of the methodology to automate code generation and the flexibility of doing so in varying programming languages and formats. Then a third schema was integrated and generated in Java with parsing and view generation capabilities. This demonstrated the ability of the methodology to handle integration of additional schemas after the initial integration and code generation. To provide more extensive testing of the methodology and SIT tool, the above integrations were repeated in various orders. The resulting global schemas were all approximately the same (some attribute and class names varied). These additional integrations are not presented in detail due to their repetitive nature to those already presented.

V. Results, Conclusions and Recommendations

Today many data stores exist with similar data in heterogeneous formats. It is often desired to use the data from one or more of these data stores with tools developed for another data store and format. Much research has been done in this area, and recently several AFIT theses have proposed methodologies for allowing data format translation and integration. All of these methodologies require large amounts of time writing code to define an integrated format and to provide tools to parse local data into the global format, create local views of data in the global format, and export global data into the local formats. The primary objective of this research effort was to identify ways to automate this integration and tool development, and to provide a methodology based on using the discovered techniques and tools.

5.1 Results

The methodology for this effort is based on developing a schema integration tool (SIT). However, the use of the tool provides automatic generation of code. The automatically generated code defines global, mapping, and local schemas. The code also parses local data into the global format, generates local views of the global data, and parses global data into local formats. The methodology focuses most of the time, effort, and cost of the integration into the development of the SIT. This allows the greatest amounts of time, effort, and cost to be expended once, with savings every time integration is performed. Also, once developed, a SIT may be used for multiple different integration efforts, resulting in even greater savings. Further, the SIT may be minimally developed with only the required capabilities for current integrations, and then incrementally extended with more capabilities as they become regularly needed. Increasing the capabilities of the SIT requires no additional training or change of techniques for the human integrators. However, as the SIT's capabilities are increased, the kinds of global schema changes requiring them to go back and adjust existing integrations will be reduced or eliminated, and more tool and code generation will become available.

74

This research also expands the classification of heterogeneity conflicts and resolutions, for use in the methodology, beyond those in the previous thesis efforts. This was done by incorporating elements from classifications developed by Kim and his associates [19, 20], and Missier and his associates [8]. Both of these classifications were developed for use while integrating relational databases, and relational databases to object-oriented databases in Kim's case. This effort took Kim's and Missier's classifications and resolutions and adapted them for use in integrating object-oriented schemas (database or otherwise) with other object-oriented schemas.

Kim and Missier's resolutions depended heavily on the use of database queries and views formed by using SQL (in some cases expanded for object-oriented use). This methodology uses mapping schemas made up of object-oriented views adapted from those defined by Guerrini and associates [15]. The SIT interactively prompts the integrator for the needed information to create the mapping views. This allows for the view techniques to be used as they are with relational databases, but does not require the use of SQL or OQL (described in Section 2.4), allowing for flexibility in whether or not local and global schemas must be associated with an OODBMS. Therefore, this methodology allows for aggregation and inheritance conflicts to be resolved by breaking them down into more simple conflicts and solving each of them, or by using the mapping schemas to build a view of the global data in the correct aggregation or hierarchy structure. These conflicts are similar to table structure and many-to-many table conflicts in relational database integration, and similarly are the most complicated and difficult to resolve conflicts, with the most impact on previous integrations. Methodologies for integration of relational database schemas have used the dual conflict resolutions to achieve great flexibility in integration, while minimizing impacts to already integrated schemas. With this methodology, that same flexibility and minimizing of impacts is available. This allows for more automation in keeping existing integrations up to date with global changes.

5.2   Conclusions

The initial focus of the research was to evaluate and identify formal methods and knowledge-based software engineering (KBSE) techniques and tools which could be used

to automate the generation of code accomplished by hand in previous AFIT theses and methodologies. It was found that by using tools based on formal methods and KBSE techniques, the integration of schemas and the automation of code generation could be accomplished without the integrators needing training in formal methods. The tools can be developed to "shield" the integrators from the formal nature of the techniques.

Very rarely will the exact same data be stored in two different local formats. Occasionally, all the data in one format is a subset of the data in the other format, but most often it is subsets of data from both local formats that is the same. Therefore, additional data, and often data processing, will usually be required before data from one local format can be transferred to another local format. Rather than adding complication to the schema integration by doing some data integration, and then having the same complexity in the end-tools to complete the data integration, it was thought to be best if schema integration and data integration were kept separate. The demonstration of the methodology in Chapter 4 seems to indicate otherwise.

Chapter 4 presented one implementation of the SIT and demonstrated the methodology by integrating battle simulation systems. During the integration process, a couple of complicated combined aggregation and hierarchy conflicts were found. Although resolutions were completed for these conflicts, the resolutions resulted in the global schema storing the same data in different formats, depending on the local format from which it came. This is true regardless of whether the conflicts were resolved by resolving the more simple conflicts making them up, or by using the mapping schemas to create local views of the data with the proper aggregation hierarchy. This requires processing to be accomplished which transforms data from one format in the global store to another before exporting to another local format. As described above, the methodology calls for this processing to be done by end-tools or end-users.

Combining the schema integration with all or part of the data integration would result in the global schemas being smaller and less complicated in situations like the above cases. The data processing could be handled by methods which parse local data into the global format and export global data to local formats. This may also remove the requirement for the end-tools and end-users to perform data processing, leaving only the need to add and

76

edit data in the global format. Therefore, this could result in simplifying both the schema and data integrations.

5.3 Recommendations For Future Work

This effort provided a methodology for integrating data stores that facilitates the automatic generation of schema definitions, view generators, and parsers. However, there is still a lot of research that could be accomplished resulting in extending the methodology to expand the code generation capabilities and address currently out of scope issues. Four main areas of focus are recommended for research, and are discussed in the next four sections.

5.3.1 Extraction of Object-Oriented Schemas. One of the major assumptions of this effort was that the local schemas are available in object-oriented formats. However, the kinds of data stores the methodology was designed to use are rarely object-oriented. Weber [38], McDonald [23], and Pearson [27] all provide techniques and methodologies for extracting object-oriented schemas from non-object-oriented data stores. Combining these methodologies with the one in this effort would provide the ability to develop tools that could take data from any local format, parse the data into a global format, perform any desired or needed operations on the data, and export the data to any other integrated format. This is the ability desired by AFRL and many others. Research needs to be accomplished to determine the compatibility of the methodologies and the best way to integrate the methodologies and tools together. Noe [24] provides a methodology for tool integration that may be valuable in accomplishing this.

5.3.2 Data Integration. As mentioned in Section 5.2, this effort and the resulting methodology are based on the separation of schema and data integration, with data integration the responsibility of end-tools and end-users. However, the methodology demonstration in Chapter 4 indicates there may be advantages to combining the schema and data integration.

77

The classification of heterogeneity conflicts presented in Section 3.3 lists attribute integrity constraints, attribute default values, and class invariant constraints among the conflict types. No resolutions are presented for these since they deal with instance data which is part of data integration. However, the constraints and default values could be supported and their conflicts could be resolved by using formal methods to define them. Further, the attribute integrity constraints, attribute default values, and class invariant constraints define what data is allowed to be stored in the attributes and classes. All of these can be entered into the schema using Z, described in Section 2.5.2, or some other formal specification language. The attribute and class constraints could then be transformed into attribute get and set methods by the transformation tool (SIT). Also, the entry of conversion methods would be simplified to supplying method preconditions and postconditions in the selected formal specification language. The SIT would then be able to transform these conditions into method bodies. This does require integrators to receive training in formal methods and use them during the integration.

Research should be completed to determine the following:

- Whether combining the schema and data integration simplifies or complicates the total integration process.

- Whether the combination is beneficial, how much of the data integration should be combined with the schema integration.

- Whether using the formal nature of the tools and techniques of the methodology to expand the methodology to include data integration will work, and whether the results are worth the formal methods training which will be required of the integrators.

5.3.3 Artificial Intelligence. This effort determined that it is unreasonable to attempt to integrate the identification and resolution of heterogeneity conflicts. Artificial Intelligence (AI) techniques are very similar to knowledge-based software engineering techniques. However, the time constraints of this effort did not allow for AI techniques to be explored. It is possible that AI techniques could be developed which would help the integrator identify heterogeneity conflicts and propose solutions for them. This would provide a partial automation of the identity and solution of heterogeneity conflicts, and

could lead to complete automation for some cases. More information about AI techniques can be found in McDonald [23].

5.3.4   Object-Oriented Database Management System (OODBMS).   Colonese [6] and Weber [38] implement the global data stores using OODBMSs. They both propose that there are benefits to using an OODBMS when processing and manipulating data in the global format. This effort uses an object-oriented format, but provides the flexibility of generating the global data store in a format usable by an OODBMS or in a format independent of an OODBMS. Providing several different code generators accomplishes this ability. Developing a code generator for use with an OODBMS was not accomplished as part of this effort due to time constraints. However, one can be developed in the same manner as the Java and C++ code generators which were developed. The generators would just output into a different surface syntax.

Two areas of research are available in this area. First, the true benefits of using an OODBMS have never been fully elaborated. Second, if an OODBMS is going to be used, what additional methods should be automatically generated by the SIT and what additional information and effort would it take to accomplish their automation. Also, in conjunction with combining the schema and data integrations, are there traits or capabilities of an OODBMS that could facilitate the combining of these efforts?

5.4   Summary

The main contribution of this effort is the development of a tool-based methodology for integrating various data stores with similar data in different formats. The methodology allows for much of the effort to be automated, including code generation for local, mapping, and global schema definitions, locals views of global data, importing local data into the global format, and exporting the global data into local format. This methodology can be used for any integration effort. However, it is most useful when the following goals are desired:

- To develop a generic format, into which data can be entered once and then used with all local formats.

- To translate data from one local format into other local formats.

Portions of battle simulation system data stores were integrated to demonstrate this methodology. This included the development of a schema integration tool, on which the methodology is based. Once developed, the schema integration tool may be incrementally expanded and used for many different integration efforts.

Appendix A.   Sample Implementation of the Schema Integration Tool

The methodology presented in this effort is tool-based. The use of the transformation-based SIT allows for the following:

- Automatic conversion of a local schema into a global or mapping schema.

- Automatic conversion of SIAT members' integration and aggregation mapping inputs into the definition of attributes and methods.

- Automatic generation of code.

- The flexibility to use commercial products with end-tools, without being reliant on their continued use.

- The flexibility to incrementally increase the ability to handle more complex heterogeneity conflicts, without having to change the way the SIAT members' apply the methodology.

In other words, it is the use of the SIT that allows the methodology to meet the primary goals of automation and flexibility.

The methodology focuses most of the time, effort, and cost of the model integrations into the SIT's development. The development of the SIT also requires a lot of knowledge and understanding of transformation-systems. This appendix provides information on the SIT implementation used in Chapter 4, to serve as a template for future SIT designs. The first two sections describe the main components of the SIT. The appendix ends with a description of the transformations used by the SIT and the technique used to implement them.

A.1   Use of the AWSOME AST

As discussed in Section 4.3.1 the SIT is based on the AWSOME transformation system. The actual AWSOME system is still under development. However, the AWSOME wide-spectrum AST has been fully developed. The AWSOME AST stores a hierarchical representation of each object model being manipulated by the transformation system. A simplified logical diagram of the AWSOME AST is shown in Figure 18. The actual

AWSOME AST is described by Cornn [7]. Saving the elements of the object model in the AST allows for transformations to be developed that transform the model components based on what type of AST node they are. For example, a transform that generates Java code would know to create a file each time it encountered a Class node.
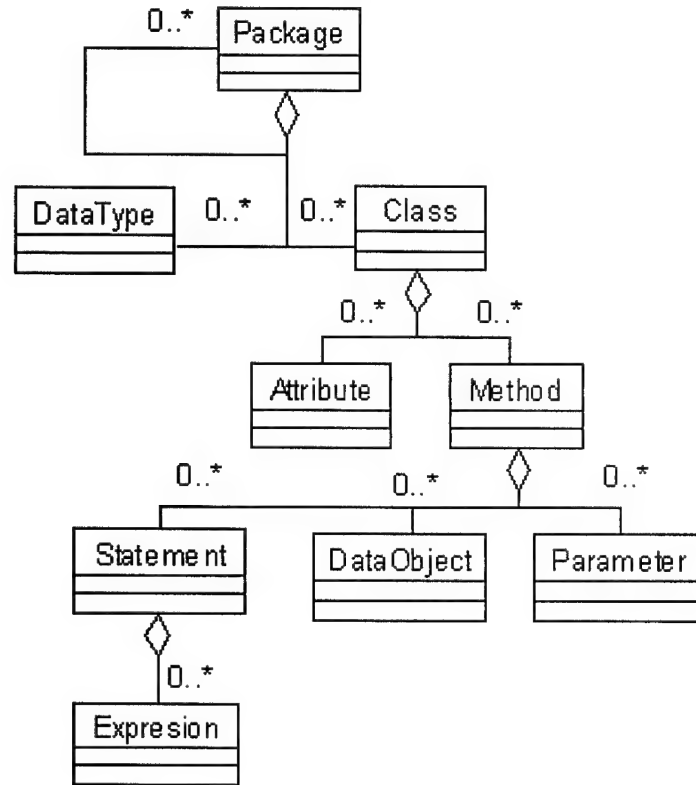
Figure 18.   Simplified Logical View of AWSOME Wide-Spectrum AST

A.2   Use of Virtual Schemas

Saving the object models in the AWSOME AST puts them in a format easy for the tool to perform transformations on. However, the user will be integrating the objects of the model, and needs to see the data as objects and not in the format of the AWSOME AST. This *object* view of the model data in the AWSOME AST is provided using virtual schemas. Virtual schemas and object-oriented views are described in Section 2.4. The packages and classes of the model are represented by object-preserving views. The model's

aggregation and inheritance hierarchy are both shown. Since the classes shown are views, changes to an object from one view are made to the corresponding AWSOME AST nodes, and then displayed by all the views of those nodes. This allows the user to view the objects in many formats and make changes in any of the formats, without causing inconsistent data or views of the data.

The SIT uses Java Swing components to display the object models graphically. The SIT displays the aggregation and inheritance hierarchies using a Java Swing *JTree*. A *JTree* will only display objects that implement the proper interface (*TreeNode*). The class and package views extend the *DefaultMutableTreeNode* class which implements all of the necessary interfaces. Using these object-oriented views allows for all additional display methods and attributes to be added without needing to change the AWSOME AST. This prevents the AWSOME AST from being arbitrarily complicated by the addition of tool specific requirements.

The integration of local models into the global model is accomplished by creating a mapping schema which defines how the data in the local model maps to the data in the global model. Mapping schemas are a special form of virtual schemas developed for this effort. Mapping schemas are described in Section 3.1.4, and an example is described in Section 3.2, Step 3. Much of the mapping is defined within the methods of the mapping schema. However, the methods are not entered directly by the users; they are developed by the SIT performing transformations on the users' input. Also, the format of the methods neither allows the SIT to detect which mappings are effected by changes to the global schema, nor perform the required updates. Therefore, the mapping schema views contain mapping "integration dictionary" classes which hold the mapping data. These integration dictionary classes are treated as base classes, not views. Therefore, there is only one of each of the integration dictionary classes for each class in the mapping schema, and all of the views for the class reference the same integration dictionary classes. Three integration dictionary classes are used by each mapping class view in the virtual schema of the mapping schema:

- Integration maps which contain the local and global attribute names and types of the mapped attributes. It also contains the names of any conversion methods used in mapping. A variable identifies whether the attribute mapping is accomplished with direct one-to-one mapping, one-to-one mapping with conversion methods, or many-to-many mappings using conversion methods.

- Aggregation maps which contain the name of the global class linked to the mapping aggregate class, the name of the global class linked to the component mapping class, and the mapping that shows how to get from the aggregate global class to the component global class.

- A vector of global classes linked to the mapping class.

- A vector of conversion methods used by the mapping class.

The global virtual schema also makes use of one integration dictionary class with each of its global class views. This class identifies the name of the mapping classes and the name of their mapping schemas, that have attributes mapped to the global class' attributes. This allows the SIT to identify which mapping schemas are effected by each change to the global schema. The Java class declarations for the mapping aggregation and integration, and the global integration dictionary classes are shown in Figures 19, 20, and 21 respectively. For conciseness, the methods of these classes are not shown.

```
public class GMVAggregationMap extends GUIObject
{
        protected String aggGlobalClass = null;
        protected String compGlobalClass = null;
        protected String aggMap = null;
        protected String collTypeName = null;
        //set null for non-collection component attributes
}
```

Figure 19.   Mapping Schema Aggregation Map Dictionary Class

```
public class GMVIntegrationMap extends GUIObject
{
        protected String attrName = null;
        protected String attrType = null;
        protected String globalClass = null;
        protected String globalAttrName = null;
        protected String globalAttrType = null;


        protected String mapType = null;
        // Attr, AttrMethod, Method are possible choices
        protected String convGetMethod = null;
        protected String convSetMethod = null;
}
```

Figure 20.   Mapping Schema Integration Map Dictionary Class

```
public class GBLIntegrationMap extends Object implements Serializable
{
      protected String attrName = null;
      protected String attrType = null;
      protected String mappingPackage = null;
      protected String mappingClass = null;
      protected String mappingAttrName = null;
      protected String mappingAttrType = null;
}
```

Figure 21.   Global Schema Integration Map Dictionary Class

A.3 Transformations and Visitor Classes

Transformations are operations that traverse the tree, and modify the data in the tree based on the class type of each tree node. There are many transformations used in the SIT:

- Copying a local schema.

- Converting a local schema into an equivalent global schema with the appropriate mapping schema in place and providing all integration dictionary classes necessary to integrate the local schema with the global schema.

- Converting a local schema into an equivalent mapping schema, including creating all local class reference attributes.

- Removing all global and mapping class methods, and mapping global class reference attributes. This accomplishes undoing a "finalize integration".

- Converting the integration dictionary global class vector of each mapping class view into global class reference attributes of the mapping class.

- Creating *get* and *set* methods for each of the attributes in all classes of the global, mapping, and local schemas.

- Converting the integration map integration dictionary classes of the mapping class views into *get*, *set*, *import*, and *export* methods of the mapping class. The *get* and *set* methods access the appropriate attributes of the linked global classes. The *import* and *export* methods access the appropriate attributes of the local classes.

- Adding the conversion methods from the integration dictionary conversion method vector to the mapping class.

- Converting the integration dictionary aggregation maps into transformation methods within the mapping schema classes. These methods traverse the global, mapping, and local trees to parse data between the local and global schemas (both directions) and create local views of global data.

- Generating code in a selected programing language, currently C++ or Java.

The virtual schemas of the global, mapping, and local schemas are only meant for use with the SIT. Therefore, the transformations directly involving the virtual schemas are accomplished using methods defined in the package and class views. The AWSOME AST is the basis of the AWSOME transformation system and has many tools and capabilities being developed as part of the AWSOME system that use the AWSOME AST. Therefore, it is undesirable to complicate the AWSOME AST with transformation methods specific to each of the specific tools that use it. Also, due to the many tools being developed, the AWSOME AST classes would be constantly changing and requiring each of the tools to be constantly updating their copies of the classes and recompiling. The SIT is not required to remain compatible with other tools developed for AWSOME, but there are many advantageous tools being developed that could increase the model development, storage, and semantic checking abilities of the SIT. One good example is the COIL [13] parser being developed which was able to be used to parse in text files containing conversion methods. Once completed, this parser would provide a way to link the SIT with other modeling tools such as Rational Rose. Rational Rose is commonly used to develop models in the DOD and commercial industry. A link between Rational Rose and the SIT would provide an easy way to convert these existing models into the AWSOME format.

To allow for development of operations and transformations against the AWSOME AST without impacting its format, the AWSOME AST was developed for use with *Visitor* [4, 11] classes. *Visitor* classes contain one *visit* method for each class (*node*) in the tree. Each *visit* method is the method that would have normally been added to the *node* class definitions of the AST. Each *node* in the AST is given a method, *acceptVisitor*, that takes a *Visitor* as a parameter and calls the appropriate *visit* method of the *Visitor*. All *Visitor* classes *implement* a common *Visitor* interface. This allows the same *acceptVisitor* methods to be used for all *Visitor* classes. Therefore, any number of operations and transformations can be developed to operate against the AST without having to change the *node* class definitions. Both the *acceptVisitor* and *visit* methods have *Objects* as parameters and return values. *Objects* are Java classes that all other Java classes inherit from (*extend*). This allows data to be passed into and returned from the *Visitor*. A sample Java *acceptVisitor* method is shown in Figure 22. A sample Java *visit* method for copying a method is shown

in Figure 23. The biggest disadvantage to using *Visitor* classes is that they must all be updated whenever the AST changes.

This appendix presented the main concepts and techniques used to develop the prototype SIT implemented for this effort. The two main tools that form the basis for the SIT were presented. The transformations used by the SIT and the main technique used to implement them were also discussed.

```
public Object acceptVisitor (WsVisitor visitor, Object data)
{
      return visitor.visit(this, data);
}
```

Figure 22.   Sample *acceptVisitor* Method

```
public Object visit(WsMethod node, Object data)
{
      WsMethod out = new WsMethod();
      out.setWsPrivate(node.getWsPrivate());
      out.setWsClassMethod(node.getWsClassMethod());
      if (node.getWsMethodSubprogram() != null)
            out.setWsMethodSubprogram((WsSubprogram)
                  node.getWsMethodSubprogram().acceptVisitor(this, null));
      return out;
}
```

Figure 23.   Sample *visit* Method

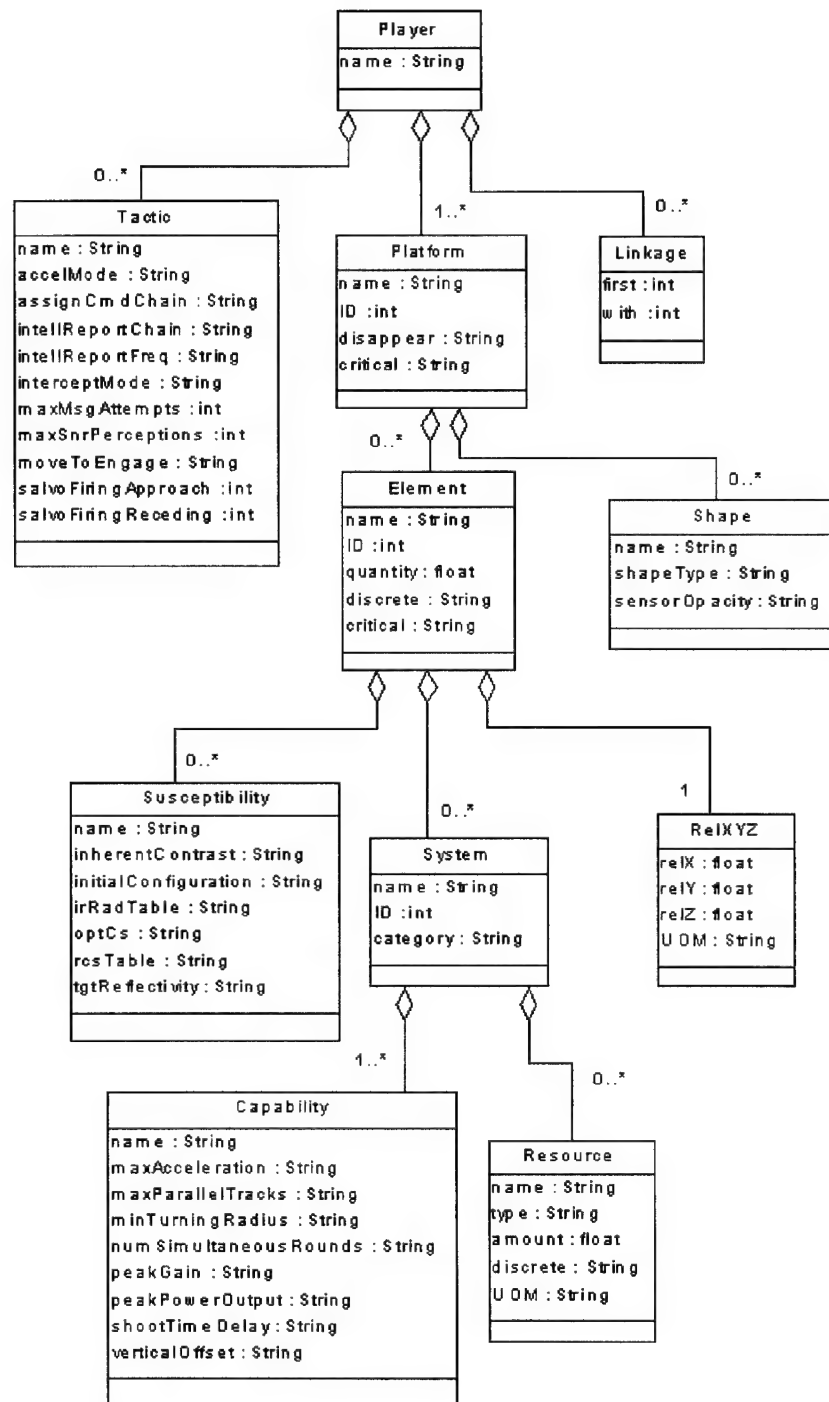*Appendix B.   Simulation Object Models Used for Integration*

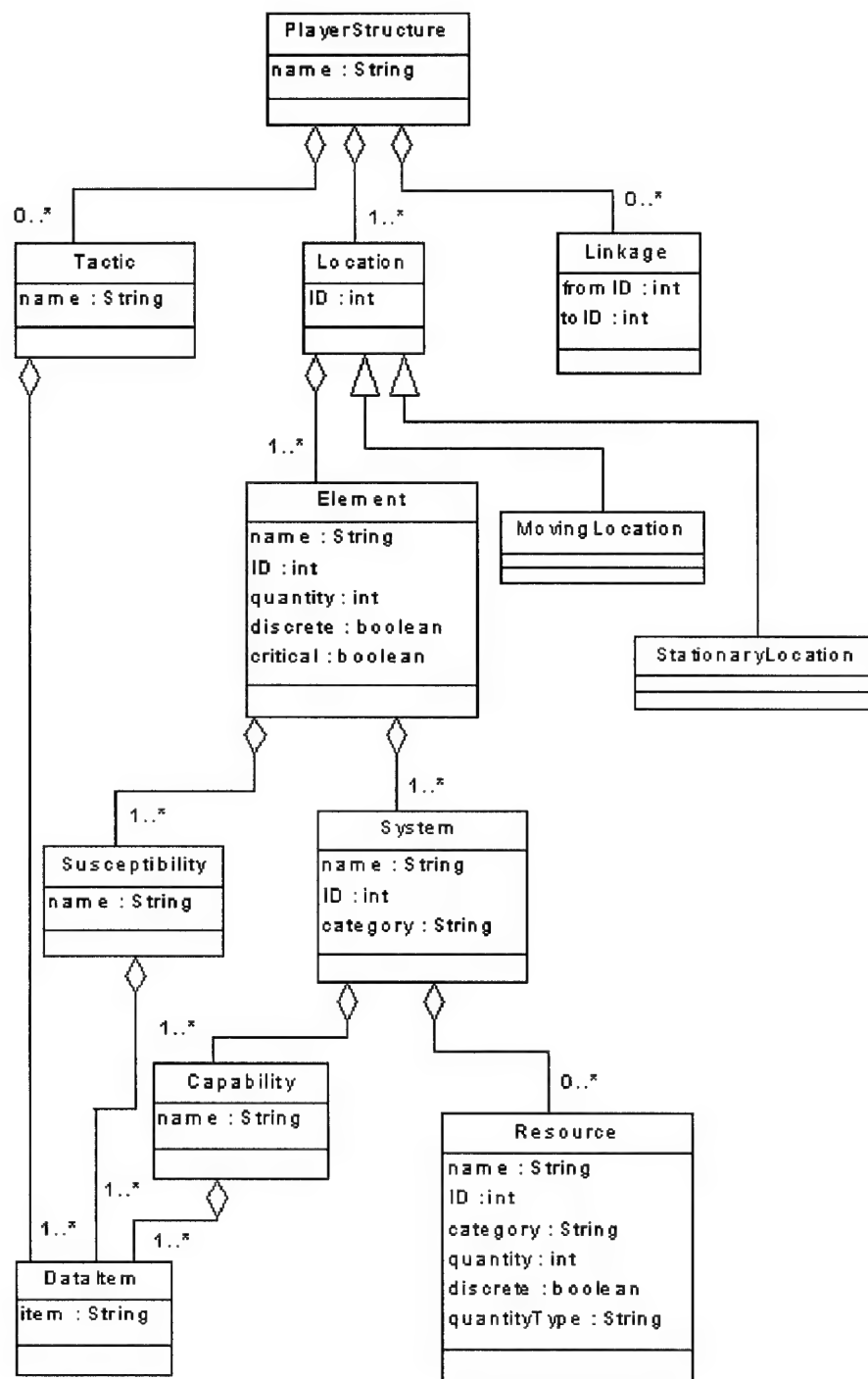Figure 24.   SWEG Object Model [38]

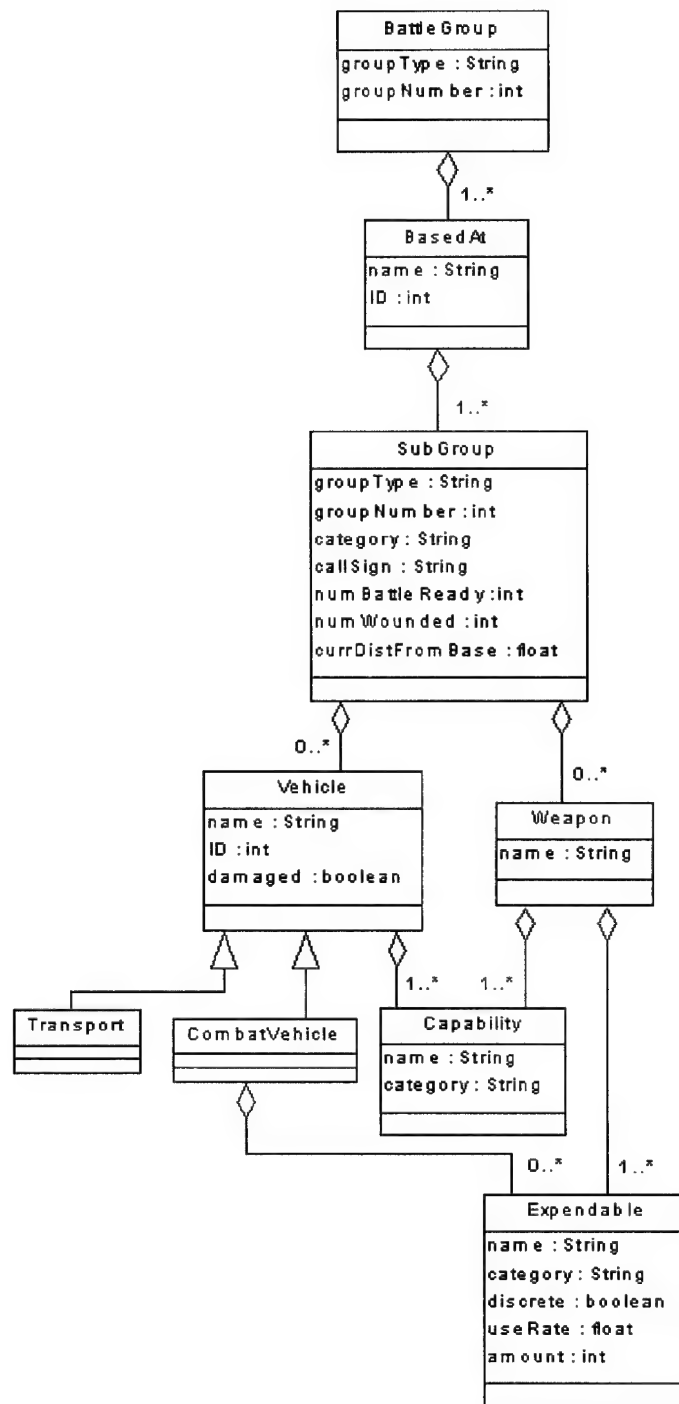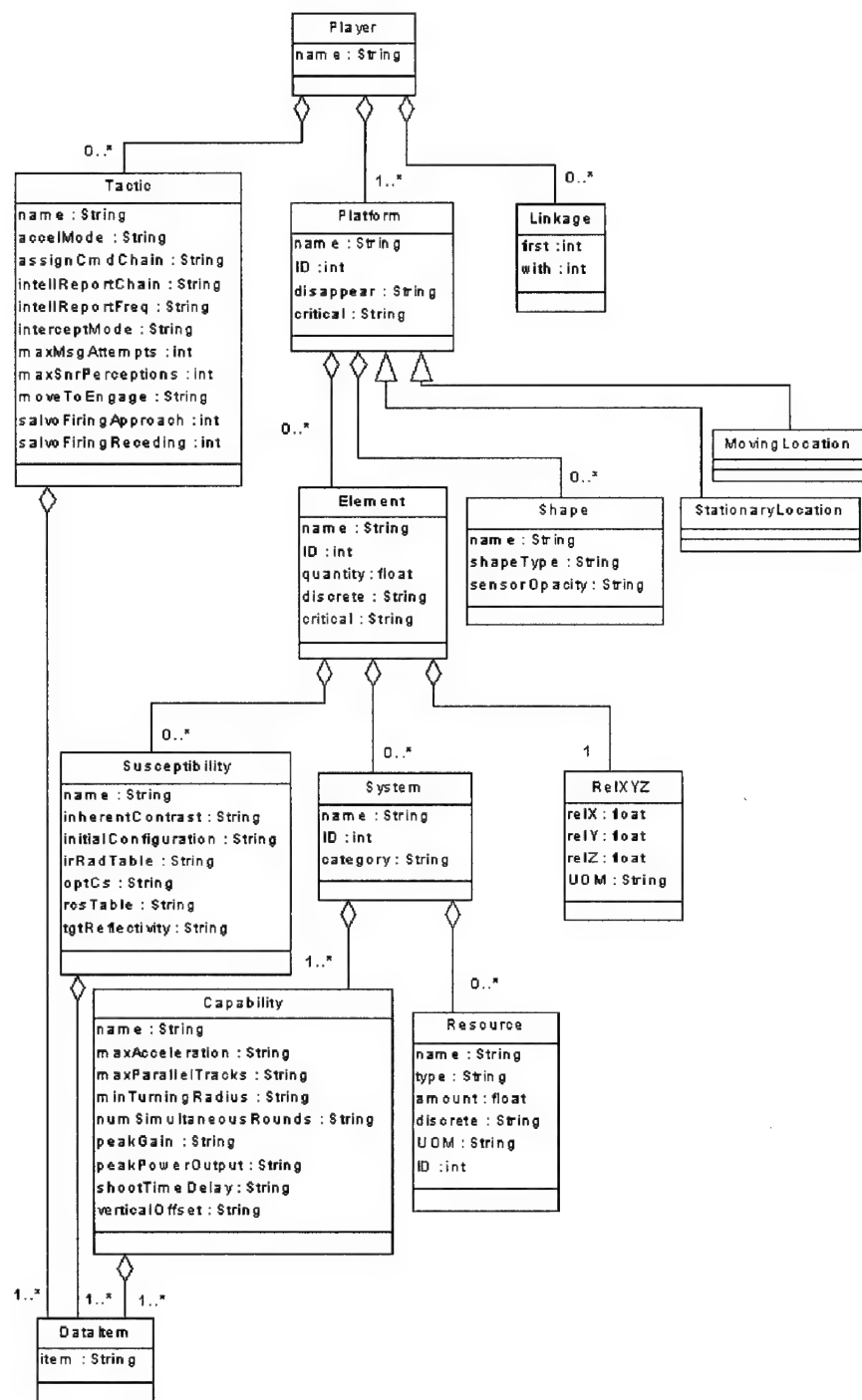Figure 25.   SUPPRESSOR Object Model [23]

Figure 26.   ASHSIM Object Model

Figure 27.   Global Schema for Integrated SWEG and Suppressor
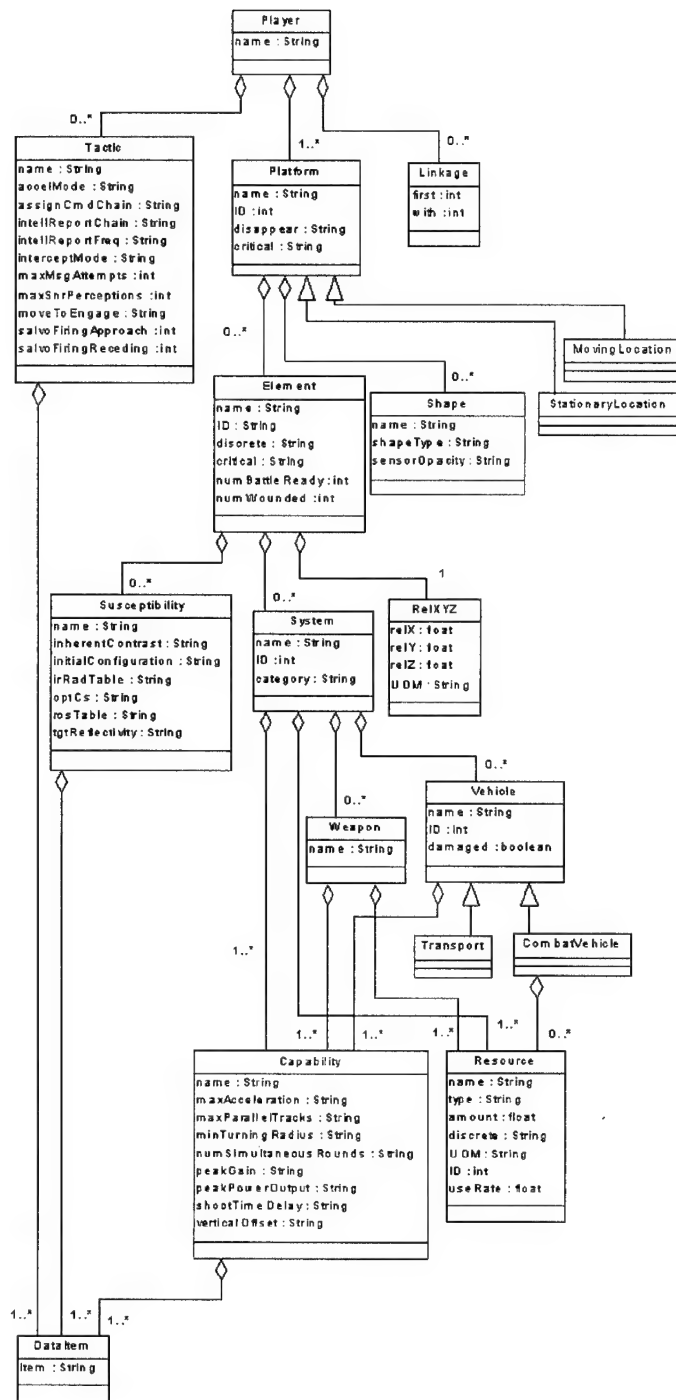
93

Figure 28.   Global Schema for Integrated SWEG, Suppressor, and ASHSIM

Appendix C.   Detailed Conflict Resolutions for Integration of SWEG, Suppressor, and ASHSIM

Chapter 4 presents a demonstration of the methodology developed in this effort. A SIT implementation, described in Appendix A, was used to integrate the SWEG, Suppressor, and ASHSIM sensor-based engagement-level simulation models. However, the details of the conflict resolutions and conversion methods used during integration are left out. This appendix provides those details. SWEG was converted to the global model, and therefore resulted in no conflicts. The specific details of the Suppressor integration conflicts are present first, followed by the ASHSIM conflict details.

C.1   Integration of Suppressor

Comparing the Suppressor schema to the global schema reveals many naming and attribute domain definition conflicts. There are also several class definition conflicts and a few schema conflicts. The methodology calls for examining the schema conflicts first since they may overlap or break down into other types of conflicts. First the conflicts and resolutions are described, and then the effects of the conflicts on the mapping step are described.

C.1.1   Schema Conflicts.   One inheritance hierarchy conflict was found. Suppressor has a class *Location* with two subclasses: *MovingLocation* and *StationaryLocation*. The corresponding class in the global schema is the *Platform* class, which has a different name and no subclasses. This conflict broke down into two missing class conflicts and a naming conflict. The missing classes needed to be added as subclasses of a global class that is a leaf node in the global inheritance hierarchy. Therefore the missing classes were able to be added with the appropriate superclass, without causing any additional conflicts. The naming conflict was handled with the rest of the naming conflicts through the mapping process. One aggregation conflict was also found. This conflict and its resolution are described in Section 4.4.

C.1.2   Class Definition Conflicts.   Only one class definition conflict was found; a missing attribute conflict. The local *Resource* class had an *ID* attribute, which the global *Resource* class did not have. This was easily resolved by adding the missing *ID* attribute to the global *Resource* class.

C.1.3   Attribute Domain Definition Conflicts.   Several data type conflicts were identified, but no other attribute domain definition conflicts were found. The data type conflicts were between integer (int) and boolean data types in the local schema corresponding to float and string data types respectively in the global schema. The float type is able to store all elements of the integer domain plus more, and the string domain is able to store true, false, and more. Therefore, the global schema already contained the most broad data types, and needed no modifying. The conversions between data types were handled during the mapping step of the methodology.

C.1.4   Naming Conflicts.   The naming conflicts were solved trivially through the regular mapping process. Therefore, they are not further discussed in this effort.

C.1.5   The Map Step and Conversion Methods.   The compare, prepare, and resolve step identified what classes and attributes should be mapped to each other and what heterogeneity conflicts prevented the mapping. Further, resolutions for the conflicts were also identified. Once identified, all resolutions were accomplished, except for naming conflicts and those involving data type conversion and aggregation mapping. As mentioned above, naming conflict resolutions are trivial to resolve and need no discussion.

The tasks of the map step are described in Section 4.5. However, when attributes were mapped whose domains did not match, the SIT required either automatic type casting or conversion methods to be used to convert between domains. Data type conversions were needed for this integration. They were integer to float, float to integer, boolean to string, and string to boolean. Automatic type casting was selected for float to integer and integer to float. The Java and C++ languages support both of these, and are currently the only languages for which the SIT has code generators. Conversion methods were selected for

boolean to string and string to boolean. The SIT has several options for defining conversion methods:

- A text file can be provided, which the SIT reads in and parses into the proper AWSOME AST format for a subprogram.

- Several common conversion methods are provided in AWSOME AST format and may be selected.

- An expression can be provided, which the SIT parses and uses to build a method.

For this integration, the boolean to string conversion method was supplied by file and parsed in. Although a full COIL to AWSOME parser was not available, a partial one was available and able to handle the simple subprogram shown in Figure 29. The string to boolean conversion method was selected from predefined conversion methods. The two different methods were used to demonstrate the different options. Both conversion methods could have been accomplished either way.

```
function booleanToString(booleanValue : in boolean) : String is
begin
     if booleanValue = true then
          booleanToString := "true";
     end if;
     if booleanValue = false then
          booleanToString := "false";
     end if;
     booleanToString := "false";
end;
```

Figure 29.   A Simple Boolean to String conversion Subprogram

C.2   Integration of ASHSIM

ASHSIM was the last local schema to be integrated. It was specifically designed to cause some more interesting heterogeneity conflicts. Since many of the heterogeneity con-

flicts were similar to those found during the integration of Suppressor, only the highlights and areas of special interest are presented.

C.2.1   Schema Conflicts.   One schema conflict was identified, a combined aggregation and hierarchy conflict. Although the conflict is described in Section 4.4, the details of the resolution used are not. In the global schema the *Element* class has a set of *System* class instances, each with a set of *Capability* class instances and a set of *Resource* class instances as components. In the ASHSIM schema there is a *SubGroup* class that corresponds to the global *Element* class. The *SubGroup* class has a set of *Vehicle* class instances and a set of *Weapon* class instances as components. These classes both correspond to the global *System* class and both sets should map to the single set of *System* class instances in the global schema. This conflict was further complicated by *Vehicle* having two subclasses. A closer look at the classes show that both the *Vehicle* and *Weapon* possible class instances are a subset of the possible *System* instances. The basis for the subsets is the value of the *category* attribute of the global System class. The *Vehicle* and *Vehicle* subclass instances correspond to *System* instances with *category* values of vehicle, transport, or combat vehicle. The *Weapon* class instances correspond to *System* instances with a *category* value of weapon. *Vehicle* and *Weapon* have a subset of the *System* attributes, except *Vehicle* adds a boolean *damaged* attribute.

Several ways exist to resolve this conflict. For this effort the conflict was resolved by adding a component set of *Vehicle* instances and a component set of *Weapon* instances to the global *Element* class, in addition to the existing component set of *System* instances. Then normal aggregation mapping was used to solve the rest of the conflict by mapping to the *Vehicle* and *Weapon* sets, but not the *System* set. This does require end-tools to process data between component sets based on which local format is to be displayed or receive exported data.

C.2.1.1   Class Definition Conflicts.   There were several class definition conflicts. However, they were various kinds of attribute isomorphism conflicts, except one, which was a combination of an attribute composition conflict and an attribute isomorphism conflict. The local *SubGroup* class having a *currDistFromBase* attribute that represents the

same information as the global *RelXYZ* component class causes the combination conflict. It is also an attribute isomorphic conflict because *RelXYZ* identifies the location by three attributes representing the three coordinates for three dimensional space. The attribute *currDistFromBase* is a single number representing the distance of the subgroup from its base location. The global schema has the class so it can store the most data. Therefore, solving this conflict just involved mapping and is discussed in Section C.2.2.

Three more attribute isomorphism conflicts were found that are worth mentioning. Both the *BattleGroup* and *SubGroup* local classes have a *groupType* string attribute and a *groupNumber* integer attribute. The attributes correspond to the *name* string attribute in the *Player* and *Element* global classes. The *groupType* string concatenated with the string representation of the *groupNumber* integer represents the equivalent data stored in the *name* string attribute of the global classes. The local schema does not provide additional information; it just breaks it up more. Since ASHSIM was the only local schema being integrated to need this, it was decided to not change the global schema, and conversion methods were used with the local attributes. These conversion methods are discussed more in Section C.2.2.

The final attribute isomorphism was caused by the sum of the *numBattleReady* and *numWounded* attributes of the local *SubGroup* class mapping to the *quantity* attribute of the global *Element* class. In this case, more semantic meaning was being stored in the local schema. Therefore, the *numBattleReady* and *numWounded* attributes were added to the global schema and the *quantity* attribute was removed. This caused new attribute isomorphic conflicts for existing mapping schemas. When the SIT prompted for the new mapping to replace the old mappings to *quantity*, an option was presented to retain the *gets* and *sets* for *quantity*, but base them on using equations involving other attributes. This option was selected. The SIT then queried for the equations to be used for the *get* and *set* methods. The equation *numBattleReady + numWounded* was used for the *get* method, and the equations *numBattleReady := newValue* and *numWounded := 0* were used for the *set* method. The SIT was able to parse these expressions into the proper statement formats for the methods.

C.2.1.2 Attribute Domain Definition Conflicts. Several data type conflicts were already discussed during the Suppressor integration. However, none of those data type conflicts required a change to the global schema. For this integration, the local *SubGroup* class has a *callSign* string attribute that corresponds to the global *Element* class' *ID* integer attribute. Since a string data type has a broader domain than the integer type, the global *ID* was changed from an integer data type to a string data type. This caused data type conflicts with previously integrated mapping schemas. The SIT detected this and queried for a conversion method or automatic type casting. In this case conversion methods for integer to string and string to integer were provided by the SIT and were selected. Once the selection was made, the SIT handled updating all attribute maps in the existing mapping schemas.

C.2.2 Step 3: Map. The attribute and aggregation mapping for ASHSIM was a little different from Suppressor's. During the compare, prepare, and resolve step, heterogeneity conflicts were identified with solutions involving mapping. Proceeding with the mapping as usual completed the resolutions except in three cases.

The *currDistFromBase* attribute of the local *SubGroup* class needed to be mapped to the *relX* and *relY* attributes of the global *RelXYZ* class. A conversion method based on an equation was used to convert the two attribute values into the single attribute value. This appendix has already discussed using conversion methods in conjunction with attribute maps. However, this time there was a one to many mapping, which does not work with attribute maps. The SIT provided a "link using methods" option that allowed two conversion methods to be used in place of the attribute map. This choice was selected. The needed equation to go from *relX* and *relY* to *currDistFromBase* is $(relX^2 + relY^2)^{1/2}$. The SIT provided an option for an expression method that prompts the user for an expression and builds a *get* method, which returns the value of the evaluated expression. This option was selected for developing the *getCurrDistFromBase* method. The SIT also provided an option for building a *set* method for a mapping attribute that allows values to be set for multiple global attributes. This option was selected for the *setCurrDistFromBase* method. The method was developed to set the *relX* value to the desired *currDistFromBase*

attribute and the *relY* attribute to zero. These values will return the proper value for the *getCurrDistFromBase* method and were arbitrarily selected from the possible valid choices.

The local classes *BattleGroup* and *SubGroup* both have *groupType* and *groupNumber* attributes which concatenated map to the *name* attributes of the global classes *Player* and *Element*. The SIT provided a conversion method that takes a string and an integer and concatenates them with a space between the values. The SIT also provided two other conversion methods which both accept a string as input. One returns the string up until a space, while the other returns the rest of the string after the space as an integer. These three conversion methods were selected for use with the attribute mappings.

This appendix provided detailed information on the conflict resolutions and conversion methods used to integrate the SWEG, Suppressor, and ASHSIM data model integrations. The flexibility of the methodology would have allowed many of the conflicts to be resolved differently. The resolutions were selected based on the abilities of the SIT implementation used for this integration, and the desire to demonstrate the flexibility of the methodology.

D.1   Global Player Class

```
package GblSWEG;
/**
* File      GblPlayer.java
*/

public class GblPlayer extends Object {
   protected     String name;
   protected     gblTacticsType tactics;
   protected     gblLinkagesType linkages;
   protected     gblPlatformsType platforms;


   public void setName(String newValue) {

      name = newValue;
   }

   public String getName() {

      return name;
   }

   public void setTactics(gblTacticsType newValue) {

      tactics = newValue;
   }

   public gblTacticsType getTactics() {

      return tactics;
   }

   public void setLinkages(gblLinkagesType newValue) {

      linkages = newValue;
   }
```

```java
    public gblLinkagesType getLinkages() {

        return linkages;
    }

    public void setPlatforms(gblPlatformsType newValue) {

        platforms = newValue;
    }

    public gblPlatformsType getPlatforms() {

        return platforms;
    }

}
```

```
package SUPPRESSOR;
/**
* File    PlayerStructure.java
*/

public class PlayerStructure extends Object {
    protected    String name;
    protected    tacticsType tactics;
    protected    linkagesType linkages;
    protected    locationsType locations;


    public void setName(String newValue) {

        name = newValue;
    }

    public String getName() {

        return name;
    }

    public void setTactics(tacticsType newValue) {

        tactics = newValue;
    }

    public tacticsType getTactics() {

        return tactics;
    }

    public void setLinkages(linkagesType newValue) {

        linkages = newValue;
    }

    public linkagesType getLinkages() {

        return linkages;
    }
```

```java
    public void setLocations(locationsType newValue) {

        locations = newValue;
    }

    public locationsType getLocations() {

        return locations;
    }

}
```

```
package MapSUPPRESSOR;
/**
* File     MapPlayerStructure.java
*/

public class MapPlayerStructure extends Object {
   protected     mapTacticsType tactics;
   protected     mapLinkagesType linkages;
   protected     mapLocationsType locations;
   protected     SUPPRESSOR.PlayerStructure localPlayerStructure;
   protected     GblSWEG.GblPlayer viewOfGblPlayer;


   public void setTactics(mapTacticsType newValue) {

      tactics = newValue;
   }

   public mapTacticsType getTactics() {

      return tactics;
   }

   public void setLinkages(mapLinkagesType newValue) {

      linkages = newValue;
   }

   public mapLinkagesType getLinkages() {

      return linkages;
   }

   public void setLocations(mapLocationsType newValue) {

      locations = newValue;
   }

   public mapLocationsType getLocations() {

      return locations;
   }
```

```java
public void setLocalPlayerStructure(SUPPRESSOR.PlayerStructure newValue) {

    localPlayerStructure = newValue;
}

public SUPPRESSOR.PlayerStructure getLocalPlayerStructure() {

    return localPlayerStructure;
}

public void setViewOfGblPlayer(GblSWEG.GblPlayer newValue) {

    viewOfGblPlayer = newValue;
}

public GblSWEG.GblPlayer getViewOfGblPlayer() {

    return viewOfGblPlayer;
}

public void setName(String newValue) {

    viewOfGblPlayer.setName(newValue);
}

public String getName() {

    return viewOfGblPlayer.getName();
}

public void exportName(String newValue) {

    localPlayerStructure.setName(newValue);
}

public String importName() {

    return localPlayerStructure.getName();
}
```

```java
public void initView() {
   MapTactic tempMapTactic = null;
   MapLinkage tempMapLinkage = null;
   MapLocation tempMapLocation = null;

   setTactics(new mapTacticsType());

   while (getViewOfGblPlayer().getTactics().hasMoreElements()) {
      tempMapTactic = new MapTactic();
      tempMapTactic.setViewOfGblTactic((GblSWEG.GblTactic)
         (getViewOfGblPlayer().getTactics().getNextElement()));
      getTactics().addElement(tempMapTactic);
      tempMapTactic.initView();
   }

   setLinkages(new mapLinkagesType());

   while (getViewOfGblPlayer().getLinkages().hasMoreElements()) {
      tempMapLinkage = new MapLinkage();
      tempMapLinkage.setViewOfGblLinkage((GblSWEG.GblLinkage)
         (getViewOfGblPlayer().getLinkages().getNextElement()));
      getLinkages().addElement(tempMapLinkage);
      tempMapLinkage.initView();
   }

   setLocations(new mapLocationsType());

   while (getViewOfGblPlayer().getPlatforms().hasMoreElements()) {
      tempMapLocation = new MapLocation();
      tempMapLocation.setViewOfGblPlatform((GblSWEG.GblPlatform)
         (getViewOfGblPlayer().getPlatforms().getNextElement()));
      getLocations().addElement(tempMapLocation);
      tempMapLocation.initView();
   }

}
```

```java
public void exportView() {
    SUPPRESSOR.Tactic tempTactic = null;
    MapTactic tempMapTactic = null;
    SUPPRESSOR.Linkage tempLinkage = null;
    MapLinkage tempMapLinkage = null;
    SUPPRESSOR.Location tempLocation = null;
    MapLocation tempMapLocation = null;

    exportName(getName());
    getLocalPlayerStructure().setTactics(new SUPPRESSOR.tacticsType());

    while (getTactics().hasMoreElements()) {
        tempTactic = new SUPPRESSOR.Tactic();
        tempMapTactic = (MapTactic)(getTactics().getNextElement());
        tempMapTactic.setLocalTactic(tempTactic);
        getLocalPlayerStructure().getTactics().addElement(tempTactic);
        tempMapTactic.exportView();
    }

    getLocalPlayerStructure().setLinkages(new SUPPRESSOR.linkagesType());

    while (getLinkages().hasMoreElements()) {
        tempLinkage = new SUPPRESSOR.Linkage();
        tempMapLinkage = (MapLinkage)(getLinkages().getNextElement());
        tempMapLinkage.setLocalLinkage(tempLinkage);
        getLocalPlayerStructure().getLinkages().addElement(tempLinkage);
        tempMapLinkage.exportView();
    }

    getLocalPlayerStructure().setLocations(new SUPPRESSOR.locationsType());

    while (getLocations().hasMoreElements()) {
        tempLocation = new SUPPRESSOR.Location();
        tempMapLocation = (MapLocation)(getLocations().getNextElement());
        tempMapLocation.setLocalLocation(tempLocation);
        getLocalPlayerStructure().getLocations().addElement(tempLocation);
        tempMapLocation.exportView();
    }

}
```

```java
public void importView() {
   SUPPRESSOR.Tactic tempTactic = null;
   MapTactic tempMapTactic = null;
   SUPPRESSOR.Linkage tempLinkage = null;
   MapLinkage tempMapLinkage = null;
   SUPPRESSOR.Location tempLocation = null;
   MapLocation tempMapLocation = null;
   GblSWEG.GblTactic tempGblTactic = null;
   GblSWEG.GblLinkage tempGblLinkage = null;
   GblSWEG.GblPlatform tempGblPlatform = null;

   setName(importName());
   setTactics(new mapTacticsType());

   while (getLocalPlayerStructure().getTactics().hasMoreElements()) {
      tempMapTactic = new MapTactic();
      tempTactic = (SUPPRESSOR.Tactic)(getLocalPlayerStructure().
         getTactics().getNextElement());
      tempMapTactic.setLocalTactic(tempTactic);
      getTactics().addElement(tempMapTactic);
   }

   setLinkages(new mapLinkagesType());

   while (getLocalPlayerStructure().getLinkages().hasMoreElements()) {
      tempMapLinkage = new MapLinkage();
      tempLinkage = (SUPPRESSOR.Linkage)(getLocalPlayerStructure().
         getLinkages().getNextElement());
      tempMapLinkage.setLocalLinkage(tempLinkage);
      getLinkages().addElement(tempMapLinkage);
   }

   setLocations(new mapLocationsType());

   while (getLocalPlayerStructure().getLocations().hasMoreElements()) {
      tempMapLocation = new MapLocation();
      tempLocation = (SUPPRESSOR.Location)(getLocalPlayerStructure().
         getLocations().getNextElement());
      tempMapLocation.setLocalLocation(tempLocation);
      getLocations().addElement(tempMapLocation);
   }
```

```java
      getViewOfGblPlayer().setTactics(new GblSWEG.gblTacticsType());

      while (getTactics().hasMoreElements()) {
         tempGblTactic = new GblSWEG.GblTactic();
         tempMapTactic = (MapTactic)(getTactics().getNextElement());
         tempMapTactic.setViewOfGblTactic(tempGblTactic);
         getViewOfGblPlayer().getTactics().addElement(tempGblTactic);
         tempMapTactic.importView();
      }

      getViewOfGblPlayer().setLinkages(new GblSWEG.gblLinkagesType());

      while (getLinkages().hasMoreElements()) {
         tempGblLinkage = new GblSWEG.GblLinkage();
         tempMapLinkage = (MapLinkage)(getLinkages().getNextElement());
         tempMapLinkage.setViewOfGblLinkage(tempGblLinkage);
         getViewOfGblPlayer().getLinkages().addElement(tempGblLinkage);
         tempMapLinkage.importView();
      }

      getViewOfGblPlayer().setPlatforms(new GblSWEG.gblPlatformsType());

      while (getLocations().hasMoreElements()) {
         tempGblPlatform = new GblSWEG.GblPlatform();
         tempMapLocation = (MapLocation)(getLocations().getNextElement());
         tempMapLocation.setViewOfGblPlatform(tempGblPlatform);
         getViewOfGblPlayer().getPlatforms().addElement(tempGblPlatform);
         tempMapLocation.importView();
      }

}

public void initView(GblSWEG.GblPlayer topGblPlayer) {

   setViewOfGblPlayer(topGblPlayer);
   this.initView();
}

public void exportView(SUPPRESSOR.PlayerStructure topPlayerStructure) {

   setLocalPlayerStructure(topPlayerStructure);
   this.exportView();
}
```

```
public void importView(SUPPRESSOR.PlayerStructure topPlayerStructure) {

    setLocalPlayerStructure(topPlayerStructure);
    setViewOfGblPlayer(new GblSWEG.GblPlayer());
    this.importView();
}

}
```

Bibliography

1. Anderson, Gary L. *An Interactive Tool for Refining Software Specifications from a Formal Domain Model*. MS thesis, Air Force Institute of Technology, March 1999. AFIT/GCS/ENG/99M-01. ADA361745.

2. Bertino, Elisa. "A View Mechanism for Object-Oriented Databases," 3rd *International Conference on Extending Database Technology*, 64–69 (March 1992).

3. Booch, Grady. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, 1994.

4. Cartwright, Robert. "Notes on Object-Oriented Program Design." Rice University. www.cs.rice.edu/ cork/newBook/, 1999, unpublished.

5. Cattell, R.G.G., et al., editors. *The Object Interface Standard ODMG 2.0*. Morgan Kaufmann Publishers, Inc., 1997.

6. Colonese, Emilia. *Methodology For Integrating the Scenario Databases of Simulation Systems*. MS thesis, Air Force Institute of Technology, June 1999. AFIT/GCS/ENG/99J-03. ADA364951.

7. Cornn, Gary. *An Object-Oriented Repository-based Software Synthesis System*. MS thesis, Air Force Institute of Technology, March 2000. AFIT/GCS/ENG/00M-05.

8. Elmagarmid, Ahmed, et al., editors. *Management of Heterogeneous and Autonomous Database Systems*. Morgan Kaufmann Publishers, Inc., 1999.

9. Eriksson, Hans-Erik and Magnus Penker. *UML Toolkit*. John Wiley & Sons, Inc., 1998.

10. Flanagan, Thomas and Elias Safdie. *Data Warehouse Technical Guide*. Sybase Inc, 1999. http://www.gate1.com/solutions/whitepapers/sysbase/syb_data_wh_tech_guide.html.

11. Gamma, Erich, et al. *Design Patterns*. Addison Wesley, 1995.

12. Garone, Steve. "1998 Worldwide Market for Object-Oriented Analysis, Modeleing, Design, and Construction Tools," *IDC Bulletin #16552* (July 1998). http://www.rational.com/products/rose/reviews/analysts/01.jtmpl.

13. Graham, Robert P., Jr. "Common Object-Oriented Imperative Language Language Reference Manual." Air Force Institute of Technology, Sep 1999, unpublished.

14. Graham, Robert P., Jr. "Lecture Notes Part 1 CSCE 793 - Formal Methods in Software Engineering." Air Force Institute of Technology, Spring 1999, unpublished.

15. Gueriini, Giovanna, et al. "A Formal Model of Views for Object-Oriented Database Systems," *Theory and Practice of Object Systems (TAPOS)*, *3*(3):157–183 (1997).

16. Hartrum, Thomas C. "An Object Oriented Formal Transformation System for Primitive Object Classes." Air Force Institute of Technology, March 1991, unpublished.

17. Jacobson, Ivar, et al. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.

18. Jordan, David. *C++ OBJECT DATABASES Programming with the ODMG Standard*. Addison Wesley, 1998.

19. Kim, Won, editor. *Modern Database Systems; The Object Model, Interoperability, and Beyond*. Addison Wesley, 1995.

20. Kim, Won and J. Seo. "Classifying Schematic and Data Heterogeneity in Multidatabase Systems," *IEEE Computer* (December 1991).

21. Kopka, Helmut and Patrick W. Daly. *A Guide to LaTeX* (Third Edition). Addison Wesley, 1999.

22. Lattimore, Peter J. *SWEG Release 6.4 User's Guide*. Bosque Technologies, Inc., 1996.

23. McDonald, Todd. *Agent Based Framework for Collaborative Engineering Model Development*. MS thesis, Air Force Institute of Technology, March 2000. AFIT/GCS/ENG/00M-16.

24. Noe, Penelope Ann. *A Structured Approach to Software Tool Integration*. MS thesis, Air Force Institute of Technology, March 1999. AFIT/GCS/ENG/99M-14. ADA361674.

25. Pantham, Satyaraj. *Pure JFC Swing A Code-Intensive Premium Reference*. Sams, 1999.

26. Park, Heon-Gyu. *The Development of a Scenario Translator for Distributed Simulations*. MS thesis, Air Force Institute of Technology, December 1996. AFIT/GCS/ENG/96D-22. ADA323341.

27. Pearson, Joe. "Automating Translation of Heterogeneous Schemas Into a Common Object Model (tentative)." (Will be published as an AFIT thesis), 2000.

28. Rational Software Corperation. *Rational Rose 98 Extensibility Reference Manual*, 1998.

29. Rational Software Corperation. *Rational Rose 98 Extensibility User Guide*, 1998.

30. Rational Software Corperation. *Rational Rose 98i Product Information*, October 1999. http://www.rational.com/products/rose/prodinfo/98ifeatures.jtmpl.

31. Reasoning Systems Inc. *REFINE User's Guide*, 1990.

32. Rumbaugh, James, et al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

33. Science Applications International Corporation. *SUPPRESSOR Release 5.4 User's Guide*, 1997.

34. Simulation Technology Branch. *CERTCORT Requirements Document*, 1997. Systems Concepts and Simulation Division, Avionics Directorate, Wright Laboratory, Air Force Materiel Command, Wright-Patterson Air Force Base, OH.

35. Stratton, Phillip. *A Metrics-based Analysis of Interface Usablility Improvements by Applying Intelligent Agents*. MS thesis, Air Force Institute of Technology, March 1999. AFIT/GCS/ENG/99M-18. ADB243044 Limited.

36. Teledyne Brown Engineering. *Methodology Manual EADSIM*, 1996.

37. Teledyne Brown Engineering. *User's Reference Manual EADSIM*, 1996.

38. Weber, Robert J. *Extracting a Common Object Model for DoD Simulation Systems*. MS thesis, Air Force Institute of Technology, March 1999. AFIT/GCS/ENG/99M-20. ADB243046 Limited.

Vita

Michael Ray Ashby was born in Medford, Oregon. He graduated from Orem High School in Utah in 1989. He was married to Wendy Mayson in Aug 94, while attending Brigham Young University in Provo, Utah. In 1995 he graduated with a B.A.S. in Electrical and Computer Engineering and was commissioned in the Air Force. His first Air Force assignment was to Los Angeles Air Force Base, where he worked with the Titan II and IV Space Launch Vehicles and the Defense Satellite Communications Satellite (DSCS) systems. After three years in Los Angeles, he was selected to attend AFIT for a degree in Digital Electrical Engineering. After meeting the requirements for both degrees, he chose to graduate with a Masters of Computer Engineering with a Software Engineering focus. Michael and Wendy have two boys, James and Kent, and are expecting a daughter, Kethry, in April 2000.

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to *stay within the lines* to meet *optical scanning requirements.*

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	PR	- Project
G	- Grant	TA	- Task
PE	- Program Element	WU	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with....; Trans. of....; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD	-	See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE	-	See authorities.
NASA	-	See Handbook NHB 2200.2.
NTIS	-	Leave blank.

Block 12b. Distribution Code.

DOD	-	Leave blank.
DOE	-	Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA	-	Leave blank.
NTIS	-	Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2000	Master's Thesis

4. TITLE AND SUBTITLE

TOOL-BASED INTEGRATION AND CODE GENERATION OF OBJECT MODELS

5. FUNDING NUMBERS

6. AUTHOR(S)

Michael R. Ashby, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 P Street, Building 640
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GE/ENG/00M-02

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFRL/SNZW
Attn: Mr. R. M. Foster
Air Force Research Laboratory (AFRL)
2241 Avionics Circle, Bldg 620, Suite 1
WPAFB OH 45433-7765   DSN: 785-2811, ext. 4364

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

Dr. T. C. Hartrum, ENG, DSN: 785-3636, ext. 4581

12a. DISTRIBUTION AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Today many organizations are faced with multiple large legacy data stores in different formats and the need to use data from each data store with tools based on the other data stores' formats. This thesis presents a tool-based methodology for integrating object-oriented data models with automatic generation of code. The generated code defines a global data format, generates views of global data in individual integrated data formats, and parses data from individual formats to the global formats and from the global format to the individual formats. This allows for legacy data to be translated into the global format, and all future data to be entered in the global format. Once in the global format, the data may be exported to any of the integrated formats for use with the appropriate tools. The methodology is based on using formal methods and knowledge-based engineering techniques with a transformation system and object-oriented views. The methodology is demonstrated by a sample implementation of the integration tool being used to integrate data formats used by three different sensor-based, engagement-level simulation systems.

14. SUBJECT TERMS
Object Models, Simulations, Object-Oriented, Modeling, Database, Integration, Translation, Object-Oriented Views, Interoperability, Automation, Code Generation, Heterogeneous Conflict Resolution, Formal Methods, Transformation System

15. NUMBER OF PAGES
130

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL